A Highly Effective System Clock for Energy-efficient Computer Systems

Diego L. C. Dutra, Lauro L. A. Whately, and Claudio L. Amorim
{*diegodutra, whately, amorim*}@*compasso.ufrj.br*
Programa de Engenharia de Sistemas e Computação, COPPE-UFRJ, Rio de Janeiro, RJ, Brasil

Abstract

Energy-efficient computer systems are making increasing use of processors that have multiple core units, DVFS, and virtualization support. However, current system clocks have not been usually designed to cope with the capacity of such mechanisms to decelerate/accelerate the passage of time, which increases the time drifts in the system and produces two adverse side effects. First, a reduction in the precision of the system clocks, which makes it infeasible to run applications that are dependent on precise time measurements. Second, increasing the rate of system resynchronization with an external global clock, which counteracts the attainment of a desirable energy efficiency.

As an alternative to a system clock, we propose an original virtual clock, named RVEC, with the property that the time count is strictly increasing and precise. Furthermore, we used RVEC to build a High-Precision Global Clock (HPGC) solution which was free from resynchronization for a cluster of energy-efficient computers.

Our experimental evaluation of an implementation of RVEC in both the Linux system and the OpenVZ virtualization system as well as HPGC in Linux by using a reference cluster of three energy-efficient computer systems showed that RVEC had negligible overhead and was highly precise in comparison with representative Linux system clocks. In addition, our results indicated that the HPGC synchronization solution is highly-scalable as it can synchronize up to 1,000 nodes/s and also that it can be performed offline, which can benefit energy-limited embedded systems, as well. These preliminary results suggest that RVEC and HPGC can be highly effective alternatives to the system clock and global clock,respectively, especially for EE computer systems.

Resumo

Sistemas de computação energeticamente eficientes (EE) utilizam crescentemente processadores com múltiplos núcleos, DVFS e suporte para virtualização. Entretanto, os atuais relógios de sistema não são usualmente projetados para lidar com a capacidade de tais mecanismos de desacelerar/acelerar a passagem do tempo, o que aumenta os desvios de tempo no sistema e produz dois efeitos adversos. Primeiro, a redução da precisão dos relógios do sistema, o que inviabiliza a execução de aplicações dependentes de medidas de tempo precisas. Segundo, aumentar a taxa de resincronização com um relógio global externo, contrapondo-se a alcançar uma desejada eficiência energética.

Como uma alternativa para o relógio de sistema, nós propomos um original relógio virtual, denominado RVEC, com a propriedade de contagem de tempo ser estritamente crescente e precisa. Mais ainda, nó usamos RVEC para construir uma solução de relógio global de alta precisão (HPGC) que é livre de resincronização.

Nossa avaliação experimental de uma implementação do RVEC e HPGC em Linux e do RVEC no sistema de Virtualização OpenVZ usando um cluster de referencia com três computadores EE, mostraram que o RVEC tem sobrecarga negligível e foi altamente preciso em comparação com dois relógios de sistemas representativos do Linux. No OpenVZ, RVEC permitiu Memperf, uma aplicação dependente de temporização precisa, ser migrada entre núcleos sem afetar suas medições, em contraste com os parcos resultados usando o relógio de sistema padrão. Adicionalmente, nossos resultados indicaram que a solução HPGC é altamente escalável pois consegue sincronizar até 1.000 nós/s e também que pode ser realizada offline. Estes resultados preliminares sugerem que RVEC e HPGC conseguem ser alternativas altamente efetivas para o relógio do sistema e o relógio global, respectivamente, especialmente para sistemas de computação energeticamente eficientes.

1 Introduction

In computer systems, the system clocks, such as the High Performance Event Timer (HPET) [4], allow applications and systems software to measure execution times and to implement synchronization operations on a cluster of computer systems using an external source for a global clock. However, the precision of such system clocks depends on both the amount of time drifting that has occurred (e.g., lost interrupts) in the system and the intrinsic variation in the time intervals between interrupts.

Recently, the problem of system clock precision has been augmented by energy-efficient computer systems that employ increasing numbers of processors with multiple core units, DVFS, and virtualization support. In fact, current system clocks usually have not been designed to cope with the capacity of such mechanisms to decelerate/accelerate the passage of time. Specifically, processors with DVFS can reduce the frequency of one or more core units for the purpose of energy savings, while a virtualization support mechanism allows virtual machines to migrate between core units for the purpose of load balancing. However, the operation of both mechanisms are exposed to system interrupts, which contribute to increasing the amount of time drifts and further reducing the precision of the system clocks.

Current system clocks usually have not been designed to cope with the capacity of such mechanisms to decelerate/accelerate the passage of time. Specifically, processors with DVFS can reduce the frequency of one or more core units for the purpose of energy savings, while a virtualization support mechanism allows virtual machines to migrate between core units for the purpose of load balancing. However, the operation of both mechanisms are exposed to system interrupts, which contribute to increasing the amount of time drifts and further reducing the precision of the system clocks.

For example, TSC is frequently used as a system clock solution for computer systems with a single CPU, in combination with an external source for the global time, such as an NTP Server [8]. In spite of being simple and efficient, such a combination solution might not be applicable to a multicore processor, in which each core has its own time counter and in which a system clock must address the synchronization of multiple local counters; additionally, there is an issue with interference over the passage of time that results from the use of DVFS and virtualization support.

In addition, the use of an NTP Server as a global clock offers low precision and further exposure to time drifts; thus, this alternative could become inappropriate for real-time distributed applications and parallel applications [6, 5]. Such an alternative also increases the energy consumption that results from extra messages and interrupts [7], thus diminishing the performance of the embedded systems when there is a restriction in the energy consumption. In the case of virtualization systems, a global clock is necessary for synchronizing the system clocks for the source and destination nodes in virtual machine migration; therefore, the low precision time count of an NTP Server is usually not sufficient.

As an alternative to the system clock, we propose an original virtual clock, named the RVEC, that has the property of strictly increasing and precise (SIP) [1] time counting, which resulted from the RVEC operation being protected from the time drifts of such a computer system. Furthermore, we show that the nodes in a cluster of computers using RVEC can stay synchronized globally after initially synchronizing their RVECs by using a remote synchronization client-server algorithm. As a result of this synchronization step, every node will have built a local High Precision Global Clock (HPGC), which will not only be synchronized with the HPGC of the synchronization server but will also be synchronized globally with all of the other HPGCs in the cluster. Most importantly, all of the HPGCs are free from the need for resynchronization.

We evaluated the performance of an implementation of RVEC in both the Linux system and the OpenVZ virtualization system as well as HPGC in Linux in by using a reference cluster of three EE computers, each of which uses quad-core Intel Xeon processors with DVFS and virtualization support. In comparison with *CLOCK_MONOTONIC* and *CLOCK_PROCESS_CPU_TIME* representative Linux system clocks and TSC, our results showed that RVEC was highly precise and added negligible overhead to the Linux kernel. In addition, our results indicated that the HPGC synchronization solution is highly-scalable as it can synchronize up to 1,000 nodes/s and also that it can be performed offline, which can benefit energy-limited embedded systems, as well. These preliminary results suggest that RVEC and HPGC can be effective alternatives to the system clock and global clock,respectively, especially for EE computer systems.

The main contributions of this paper are as follows:

1. Proposal of the RVEC virtual clock, as an original solution for system clock;

2. Development and experimental evaluation of an implementation of RVEC in a Linux kernel and the OpenVZ virtualization system for multicore processors with DVFS and virtualization support;

3. Using and evaluating the RVEC solution to build a high-precision global clock that is free from resynchronization in a cluster of energy-efficient computer systems.

The remainder of this paper has the following organization. Section 2 presents related work. In Section 3, we identify the potential sources of time drifts in both the Linux kernel and the OpenVZ for multicore processors with DVFS and virtualization support; additionally, we describe the organization and integration of RVEC to keep it protected from time drifts in such systems. In Section 4, we explain how we build a high-precision global clock using RVEC and a remote synchronization client-server algorithm in a cluster of such processors. In Section 5, we describe an experimental evaluation of RVEC, and we discuss our results. Finally, in Section 6, we present our conclusion and ongoing work.

2 Related work

The correction of time drifts and the maintenance of system clock precision are usually made with an NTP daemon running in a system that periodically resynchronizes with a remote NTP server [8]. However, a local computer under a heavy workload will delay the NTP daemon execution, causing time drifts to system clocks that can achieve tens of seconds [9]. In addition, such time precision maintenance does not guarantee that a system clock has the SIP property if the time intervals between consecutive readings of the system clock are less than tens of milliseconds.

The work on High-Precision Relative Clock Synchronization Using Time Stamps Counters [13] involved the development of a global clock using TSC as the base clock together with a remote synchronization algorithm that is similar to that of NTP. Owing to the direct use of TSC, such a global clock cannot work correctly with processors that have DVFS or multiple core units.

The RADClock [11, 14] is a stateless distributed synchronization system that is built on system clocks (e.g., HPET) or time counters (e.g., TSC), which provide information on global time as well as the absolute global time to the synchronization network nodes. However, RADClock depends on a daemon for periodic resynchronization; additionally, the direct use of TSC limits its applicability in multicore processors. The work in [1] used the RADClock solution to propose a timekeeping architecture for virtualization systems with the advantage of reducing the number of time violations in comparison with Clocksource Xen and NTP. However, the proposed solution had the same RADClock limitations regarding DVFS and multicore processors.

In [3], the authors proposed an auxiliary hardware synchronization network using a remote pulse generator. In a computer cluster, such a synchronization network assured that all of the cluster nodes simultaneously received the remote clock pulse and used it to update the node local clock without involving the operating system. In spite of guaranteeing the SIP property of local clocks, the proposed solution depended on dedicated hardware.

3 Strictly Increasing and Precise Virtual Clock

The main objective of an implementation of RVEC is to protect it from the time drifts of a computer system, to make it obey the strictly increasing and precise property. First, we chose TSC as the reference for the SIP time counter because it operates without interrupts and is internal to a specific core unit. As a result of the TSC choice, the operation control of both multiple core units and changes in the core unit frequency became the only two potential sources of time drifts to protect RVEC from as will be shown.

Code 1: RVEC data structure

```
struct tb{
  u64 base_counter;
  u64 age_time_ns;}
```

Our solution was to build RVEC as a virtual clock such that its structure stored two values, the value of the last reading of the TSC that was taken by the RVEC control logic and the value of the consolidated passage of time up to the instant of the RVEC's last update operation. To this end, we represent RVEC by the *struct tb* data structure shown in Code 1, where the *base_counter* data field stores the latest TSC value and *age_time_ns* keeps the consolidated time that is effected by the program code shown in Code 2 (which will be explained shortly).

Figure 1 ilustrates the time count using RVEC. At tick A, an instance of RVEC is initialized by storing the current TSC value 10 into the RVEC's *base_counter* field and the value 0 into the RVEC's *age_time_ns* field. At tick B, the core frequency is changed from 2 to 1 which causes the execution of the program Code 2 to update the values of the RVEC fields *base_counter* and *age_time_ns* to 20 and $5 * 10^9 ns$, respectively. Note that the RVEC instance cannot be read over the B-B' time interval of the program Code 2 execution.

Code 2: RVEC update procedure

```
void update_rvec(struct tb *ptb, long CoreHZ){
  aux_tsc = get_counter();
  ptb->age_time_ns += (aux_tsc - ptb->
      base_counter)/CoreHZ;
  ptb->base_counter = aux_tsc;}
```

Therefore, RVEC allows the creation of a time count abstraction for a specific process since the instant of its

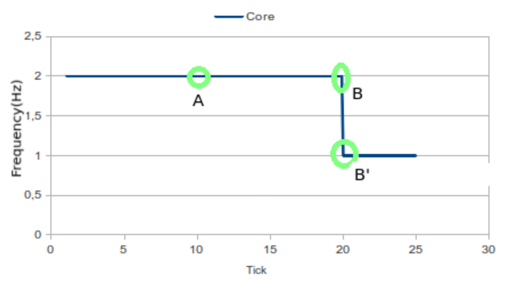

Figure 1: Time count using RVEC: initialization (tick A) and update (tick B)

creation and for other instances of the same process, with the RVEC of each process subjected to update points. Specifically, such update points are the instants in which either a core unit frequency is changed or a process migrates to another core unit. However, it is more important to insert such points in the appropriate places of an operating system so that during its execution, all of the RVEC instances become protected from potential time drifts.

3.1 RVEC in Linux

In Linux, RVEC was integrated with the sub-systems of both core management and kernel task management. For this purpose, the creation of RVEC, configured for multiple core units, was integrated into the Linux initialization procedure. Additionally, this integration allowed the system to simultaneously offer RVEC support for threads at both the kernel and user levels. Therefore, the resulting RVEC enabled the building of new RVEC instances afterward, when necessary.

Code 3: Read procedure of RVEC for a selected core

```
return uint64 rvec_core_gettime(void){
  aux_tsc = (get_counter() − ptb−>base_counter)
      ;
  return ptb−>age_time_ns + aux_tsc/get_CoreHZ
      ();}
```

We integrated RVEC into the kernel management subsystem by inserting *struct tb* into the execution queue of each core unit. In this way, such a structure is updated at the core unit initialization and whenever a core unit has its operation frequency changed. Upon a frequency change, it is necessary to guarantee RVEC correctness. This guarantee is performed by updating the fields *base_counter* and *age_time_ns* while using the program code in Code 2, which was implemented in the Linux DVFS (CPUFreq) driver. RVEC structure also maintains the consolidated time since the initialization of a core unit, which is calculated using the values of the TSC readings over the passage of time, the current frequency of the core unit, and the RVEC fields that are associated with the core unit, as shown in Code 3.

The additional control logic in the CPUfreq code extended $rvec\_cpu\_freq\_change\_pre()$ and $rvec\_cpu\_freq\_change\_pos()$, of original function code $acpi\_cpufreq\_target()$ in each core unit. Because this part of the Linux code addresses architecture-specific codes, we implemented only RVEC support in the Intel Xeon driver.

To implement the thread level RVEC in the Linux kernel, we created a *struct tb* instance for each task (thread). Specifically, upon the initialization of a new task to run in a specific core unit, the current value of RVEC associated with this core unit will be copied and stored in the *base_counter* field of RVEC that is associated with the new task. Then, an update of *base_counter* is performed whenever the task suffers migration between core units, thus guaranteeing RVEC the SIP property. Most importantly, this update control is performed in the Linux scheduling sub-system in such a way that RVEC is protected from such a potential source of time drifts.

However, an update operation will generate a call to a remote core unit, which in an overloaded Linux system can cause a kernel panic [2]. As a result, we adapted RVEC migration control to postpone updating both the RVEC *base_counter* and *age_time_ns*, as follows.

We introduced two new fields, namely *status* and *last_cpu*, into the original RVEC process structure, where the *status* indicates whether any migration occurred and *last_cpu* stores the last core unit used to maintain RVEC. Therefore, we changed the procedures to update the *base_counter* and *age_time_ns* fields at the moment at which a process checks the current value of its RVEC instance, and if the process finishes without making another access to RVEC, the data structure will not be updated unnecessarily. However, if upon an RVEC access the *status* field indicates that a migration occurred, then an access is made to RVEC in the core unit queue indicated by *last_cpu* to update the *age_time_ns* field and to store in the *base_counter* field the current value of RVEC from the current core unit queue and set the *status* to *no migration* and the *last_cpu* field to the current core.

The RVEC implementation allows different threads in the system to check their associated RVECs through the $clock\_gettime()$ system call, which goes directly into the Linux timekeeping sub-system. In this case, an application accesses that function with the clock identifier of the desired $CLOCK\_RVEC$ as the input parameter.

We plan to submit the RVEC implementation to further scalability tests, after which the RVEC source code will be freely available [3].

3.2 RVEC in the OpenVZ virtualization system

OpenVz implements virtualization at the operating system level, through resource containers [10], and also provides a controlled environment that allows us to evaluate the adapted migration algorithm with RVEC. RVEC implementation works in a shared-cluster computer environment, where all of the nodes are always found with a maximum load(PlanetLab [9]). Specifically, we introduced both the data structures and the control logic that was necessary for RVEC to work in the OpenVZ kernel. For this purpose, we modified the OpenVZ control applications running at the user level to use the new functionalities for performing migrations between cluster nodes.

We used Linux Kernel version 2.6.18 with OpenVZ patch 028stab057.2 and the previous RVEC implementation in Linux. Specifically, we changed the **struct** *ve_struct* data structure that stores the container information by inserting a **struct** *ve_struct* entry for the RVEC instance initialized at the container creation. This structure is accessed from outside of the kernel, by internal processes of the container, through *CLOCK_RVEC_VZ* clockid.

In OpenVZ, a container migration occurs between different nodes; thus, we integrated the RVEC update algorithm in the migration process. The main difference is that we now calculate the time during which a container was suspended during its migration and add this time value into *age_time_ns* at the end of its migration.

It was also necessary to integrate RVEC calls with control applications executed by the root user to manage the system containers, particularly the creation and migration process of virtual environments as well as the vzctl and vzmigrate management applications. Vzctl is an application that establishes the interface between users and the management system, which was modified to create both the migration server and the migration client, while vzmigrate was changed to use the new vzctl.

4 High-Precision Global Clock

We used RVEC to build a High-Precision Global Clock (HPGC) that is based on a client-server model, in which each client node must synchronize only once with the node chosen as the HPGC server in the cluster. Specifically, we used a client-server synchronization probabilistic algorithm [2] to determine the HPGC of a node through the equation 1.

$$RVEC_{Server}(j) \approx (RVEC_{Client_i}(j) + RTT/2), (j = 1, m(i)) \quad (1)$$

In equation 1, $RVEC_{Server}$ is the RVEC of the HPGC server node and $RVEC_{Client_i}$ is the RVEC of client node i; j is the j-th RVEC measure that is transmitted from client i to the HPGC server. The round-trip time RTT of a message is estimated by using Equation 2 and assuming that x, y are the minimum RTTs between the client i and the HPGC server; then, RTT_x and RTT_y tend to a constant value K_i after exchanging $m(i)$ messages with the HPGC server.

$$\forall_{x<>y\in\mathbb{Z}_+}(|RTT_x - RTT_y|) \to K_i \quad (2)$$

Our HPGC synchronization algorithm for a client node works basically as follows. The HPGC server, upon receiving a request from a client, will read RVEC and send its current value to the client. The client exchange requests with the HPGC server until the RTT standard deviation becomes less than a pre-defined value. At this point, the client forms its HPGC structure, which contains the minimum RTT, the server RVEC value, and the client RVEC value. From this point onward, the client can compute a global time value locally by using the function shown in Code 4.

Code 4: HPGC get_time function

```
u64 get_global_time(...){
  time_now = get_time(...) - (gC.initTimeLocal
      + gC.minimalRTT/2);
  time_now = time_now + gC.initTimeRemote;
  return time_now;}
```

Note that the above synchronization algorithm allowed us to synchronize a client node with an HPGC server node in a cluster. By assuming that the HPGC server node is the HPGC server of the cluster, it will be used by all of the other client nodes to determine its local HPGC. In the case of multiple client nodes, the HPGC server will process the synchronization requests in ascending order of request arrival time. Through the SIP property of RVECs, however, all such local HPGCs will be synchronized with the HPGC server's RVEC as well as among themselves. Moreover, the cluster nodes will stay synchronized globally without requiring any resynchronization operation over time. From now on we will refer to the syncronization algorithm execution by each node simply as the node's HPGC bootstrap.

5 Experimental Evaluation

In this section, we present the results of an experimental evaluation of RVEC and HPGC with five objectives: 1) to compare the overhead and precision of RVEC with those of representative system clocks of Linux; 2) to assess whether RVEC works correctly with DVFS operation together with and without program migration; 3) to verify RVEC's SIP property with and without migration between core units; 4) To evaluating RVEC in the OpenVZ virtualization system, and 5) to verify whether HPGC remains correct and precise without resynchronization. The experiments were conducted on a reference cluster of energy-efficient computers with 3 nodes, each having 2 quad-core 64-bit Intel Xeon E5410, 2.33 *GHz*, Linux (Ubuntu 11.10 com kernel 3.1.10), and a 1 Gb/s Ethernet switch. Additionally, we used HPET as the hardware circuit for the system clocks. Our results are presented with the average value and standard deviation given in microseconds (us) for a 99.9% confidence interval.

5.1 RVEC versus Linux system clocks

In the first experiment, we measured the RVEC overhead by using RDTSC hardware instructions to access TSC, which provides the best available precision for a system clock. Because RDTSC cannot be freely used either in a multicore system or when a core unit performs a DVFS operation, it was necessary to guarantee that all of the core units operated with the fixed maximum frequency.

This experiment was repeated 100 times and involved measuring the execution time of a *for loop* with 5,000 arithmetic instructions, with and without using a system clock past 2,500 instructions, i.e., in the middle of the loop. The *for loop* was executed 10,000 times, and in each iteration, we evaluated **TSC** using an RDTSC call, as well as **MONOTONIC**, **PROCESS_CPUTIME** system clocks and RVEC using the *clock_gettime*() system call. We included the Base clock, which measured the test program execution time without using a system clock.

Table 1: Execution time vs. Clock option

Clock option	μ(μs)	$\sigma(\mu s)$
Base	10.063	0.762
TSC	10.320	0.701
MONOTONIC	10.638	0.652
PROCESS_CPUTIME	10.571	0.627
RVEC	10.483	0.623

Table 1 shows that the TSC's average execution time of 10.320 μs represents a 2.55% overhead in relation to the Base clock. In addition, RVEC with 10.7 μs had the smallest overhead in comparison with that of the two system clocks, even when both were equally protected from system time drifts, e.g., running a Linux task scheduler. The results of RVEC's time stability indicate that RVEC can be used as an alternative high-precision clock system in Linux. Moreover, we observe that RVEC's execution overhead, like the overhead of the two system clocks, is negligible because RVEC also shares the same code path in the kernel.

In the second experiment, we compared the time precision of **RVEC**, **MONOTONIC**, and **PROCESS_CPUTIME** system clocks against that of the **TSC**.

Table 2: Time precision vs. Clock option

Clock option	μ(μs)	$\sigma(\mu s)$
TSC	9.989	0.668
MONOTONIC	10.307	1.033
PROCESS_CPUTIME	10.221	1.058
RVEC	10.262	1.047
RVEC+Mig	10.323	1.007

In Table 2, we can see that the test program's average execution time was 9.989 μs and 10.262 μs using TSC and RVEC, respectively; thus, RVEC deviated only 2.73% from TSC whereas **MONOTONIC** and **PROCESS_CPUTIME** deviated 3.18% and 2.32% from TSC, respectively. Furthermore, under the test program migration, RVEC changed only to 10.323 μs. These values of RVEC precision confirm that it can be an alternative high-precision system clock with the advantage of allowing a running application to migrate between core units without losing its time precision.

Table 3: DVFS operation effects on RVEC

Test	μ(μs)	Standard deviation
RVEC without program migration	30.652	2.331
RVEC with program migration	30.803	2.267

In the third experiment we assessed whether RVEC works correctly with DVFS operation together with and without program migration. We built our test program by replicating the arithmetic instruction block of the previous experiment, where the first and the second block was executed at maximum frequency and half of that frequency, respectively. The request for DVFS operation to change frequency was performed at the end of execution of the first block. The results are shown in Table 3. As expected the program executon time nearly tripled the value of previous experiment, and maintained fast execution under program migration; however, the standard deviation increased due to the frequency change.

Finally, we used TSC to verify whether RVEC obeys the SIP property. Note that the other system clocks were excluded from this experiment because they depended on resynchronization with an external global clock source. To assess TSC, we limited the test program process to stay within only one core unit without using DVFS, and for RVEC, we present results for both with and without process migration. This experiment was executed 100 times for each of the RVC and TSC configurations.

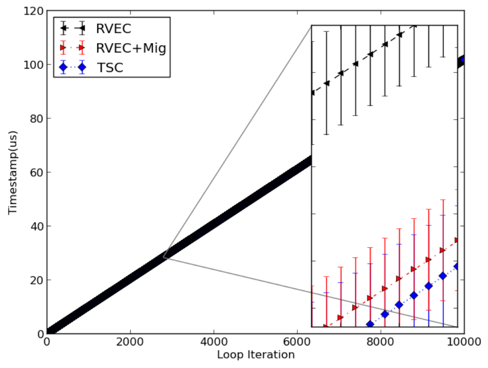

Figure 2: Assessing the RVEC's SIP property

Figure 2 presents the execution time results for TSC, RVEC, and RVEC with process migration (RVEC+Mig). It is important to note that no time violation occurred for any of the individual execution times or for the consolidated average execution times. Therefore, in the case of RVEC+mig, our results confirmed that RVEC obeyed the SIP property, as is shown more clearly in the time interval zoom of Figure 2. Surprisingly, the RVEC average execution time without migration is greater than that with migration. The explanation is that, in the latter, Linux selected the less overloaded core unit, aiming at load balancing.

5.2 Evaluating RVEC in an OpenVZ virtualization system

In this section, we evaluate the performance of RVEC in an OpenVZ virtualization system. Initially, we checked whether the passage of time measured by RVEC obeyed the SIP property for an OpenVZ container. Our experiment was to run a test program that performed 100 calls to RVEC(*CLOCK_RVEC_VZ*)using an OpenVZ container and if any of such calls returned a value that was less than or equal to the value of the previous call, the program stopped running.

We ran four experiments, each of which used a different time interval of 0 *ms*, 1 *ms*, 10 *ms*, and 1 *s* between the call to *CLOCK_RVEC_VZ* for the 1st, 2nd, 3rd, and 4th experiment, respectively. Each experiment was repeated 10 times, and at each time, 100 *CLOCK_RVEC_VZ* calls were performed. Again, the test program stopped running if the RVEC SIP property was violated when returning from any RVEC(*CLOCK_RVEC_VZ*) call.

The first experiment enabled us to investigate whether RVEC would work correctly under a stressed condition, while in the other experiments, the time intervals that we used represented traditional real-time programs. For reasons of space, we will show only the results for the 3rd experiment with 1*ms*, which we consider to be representative because, in all four experiments, the RVEC SIP property was obeyed. More specifically, the results shown in Figure 3 were obtained for the 3rd experiment over the first fifteen interactions of the experiment, confirming that the RVEC values were always increasing and precise.

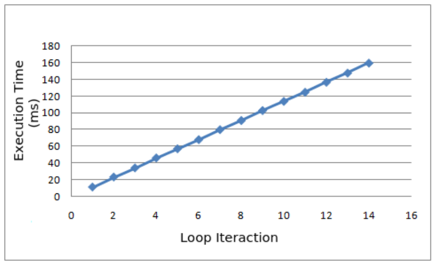

Figure 3: RVEC SIP property in the OpenVZ with 10*ms* time interval between RVEC(*CLOCK_RVEC_VZ*) calls

We validated the correctness of the OpenVZ migration algorithm that was adapted to include RVEC through a test program running in an OpenVZ container that periodically called 100 times *CLOCK_RVEC_VZ* and stop running if the RVEC SIP property was violated. The test program ran 200 times; each time, its container migrated twice, used 2 *s* and 5 *s* time intervals between *CLOCK_RVEC_VZ* calls, and always initialized the TSC of the destination node before migrating. Both of the tests concluded successfully; notably in the second test, the TSC value of the destination was always smaller than that of the source node.

5.3 Running applications that are dependent on the precise time count in OpenVZ

To understand the effects of time drifts on container migration in a practical application, we ran Memperf [12], an application that characterizes the performance of the memory hierarchy and uses the RDTSC instruction for time measurements. However, Memperf discards time measures that are less than 75% of the average time values, to avoid inserting "noise" into the time measurement set. In the first test, Memperf ran in a container that did not migrate. In the second test, the Memperfs container migrated among the computer nodes. In the third and last test, the memperfs container also migrated, but it was modified to use *CLOCK_RVEC_VZ* calls instead of the RDTSC instruction. The experiment comprised 20 Memperf runs with 100 time measurements for each test; except for the first test, the Memperf container migrated twice within each run. To avoid the cache effect, each Memperf run was preceded by 1,000 NOP instructions.

Table 4 shows that the number of discarded values was approximately 5 and 53 for the tests without migration and with migrations, respectively; in other words, the Memperfs container migrations led to 10 times more discarded measures. The reason for such a large increase in the measurement "noise" was that, during Memperf migrations, the TSC registers of the source and destination nodes were not synchronized. However, in the third test, the number of discarded values, 5, was equal to that for the first test without Memperf migration because *CLOCK_RVEC_VZ* was used by Memperf.

Table 4: Experiments with the Memperf application

Test	No. discarded values	Standard deviation
1*st* Test (RDTSC)	5.10	0.79
2*nd* Test (RDTSC + Migration)	53.15	9.16
3*rd* Test (RVEC + Migration)	5.15	0.99

Clearly, these results also support the use of RVEC as an effective system clock for applications that suffer from migrations in virtualization systems and that depend on precise time measures. Indeed, RVEC's performance reveals that it works with applications that were originally written to run with RDTSC call instructions, such as Memperf.

5.4 Global synchronization using HPGC in a computer cluster

Next, we present results on using HPGC's capability to synchronize the nodes of a computer cluster running Linux without requiring resynchronization. Our experiment used one HPGC server node and two client nodes. The HPGC server program first receive a synchronization requests from each the client and process it in a FIFO order, for each client once synchronization has started the server needed only to retransmit the messages that were sent by the clients, whereas the client algorithm is shown in Code 5, with the *usleep*() function added to avoid message flooding in the network.

Code 5: HPGC – the client algorithm

```
void exp04_Client(...){
  for (i = 0; i < 10000; i++) {
    usleep(1);
    time1 = clock_gettime(clk_id);
    sendto(...);
    recvfrom(...);
    time2 = clock_gettime(clk_id);
    *(results+i) = (time2 - time1)/2;}}
```

We repeated the experiment 200 times and measured the necessary number of messages for the variations of $RTT/2$ to be smaller than $K = 500$ *ns*, which we chose to build HPGC. The HPGC server first initiated node **A** before **B**, computing the $RTT/2$ from nodes **A** and **B** to the HPGC server at times equal to 44.51 μs and 44.47 μs, respectively. In relation to the synchronization bootstrap, each of the two nodes **A** and **B** exchanged 481 messages with the HPGC server in less than 1 *ms* using 1 μs time interval between each client-server interaction and 3 messages per interaction. After the HPGC synchronization bootstrap, each of nodes **A** and **B** used its HPGC to send 2,000 timestamp messages to the HPGC server. Figure 4 plots the global timestamps of the messages sent by **A** and **B**. We can see in this Figure that node **A** always ran first and sent the n-th message with a smaller HPGC value than that of HPGC of the n-th message from node **B**, which confirmed that the three cluster nodes stayed synchronized globally.

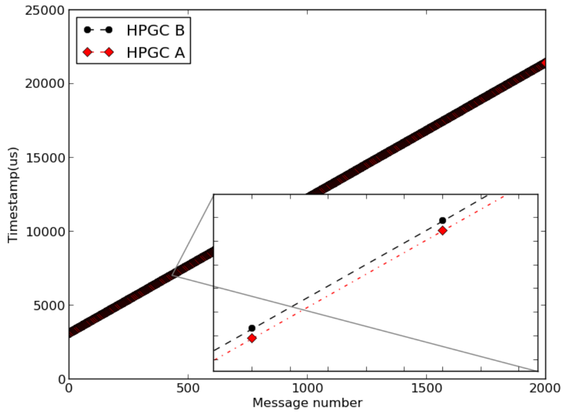

Figure 4: global synchronization using HPGC

5.5 Theoretical scalability of HPGC algorithms

In the case of a cluster of computer nodes interconnected by a non-blocking network, the HPGC synchronization algorithm can theoretically offers an extremely scalable global synchronization solution. Assume that any client node after its synchronization bootstrap can turn into an HPGC server of another group of client nodes which are waiting for performing its synchronization bootstrap. Equation 3 indicates that a 2-level HPGC algorithm can synchronize more than 500,000 nodes whereas the expressions 4 show that a totally hierarchical HPGC algorithm can synchronize 10^{301} nodes in one second, by assuming a cost of 1 ms per node' synchronization bootstrap.

$$1000 + 999 + 998 + ... + 1 \Rightarrow \sum_{n=1}^{1000} n = 500,500 \tag{3}$$

$$\left.\begin{array}{l} t = 0 \Rightarrow 1\ node \\ t = 1 \Rightarrow 2\ nodes \\ t = 2 \Rightarrow 4\ nodes \\ \vdots \\ t = 1000 \Rightarrow 2^{1000} \simeq 10^{301}\ nodes \end{array}\right\} \tag{4}$$

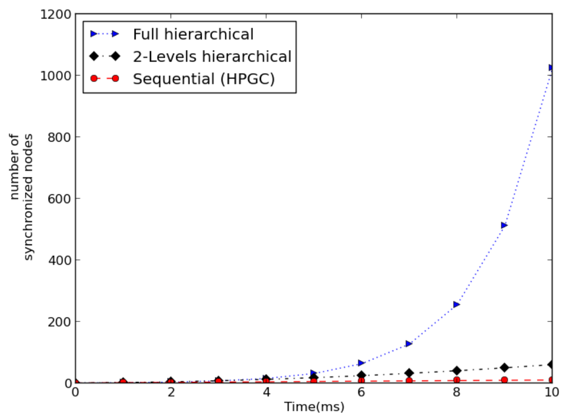

Figure 5: Theoretical scalability of three HPGC algorithms

In practice, a 1,000-node cluster can stay synchronized globally within 1 second by requiring each of the nodes to initially run the HPGC bootstrap. In contrast, an NTP solution offers much lower precision as well as it requires each of the 1,000 nodes to resynchronize every 10 seconds with the NTP server by exchanging 4 messages/resynchronizaton. Another significant advantage of the HPGC bootstrap is that it can be performed offline, which also can benefit energy-limited embedded systems such as wireless sensor networks. Figure 5 illustrates the scalability results for the three HPGC algorithms.

6 Conclusion

System clocks are not usually designed to cope with time drifts that are generated by processors with DVFS, virtualization support, and multiple core units used in energy-efficient computer systems. However, accumulating time drifts in a system can reduce the precision of system clocks, preventing correct execution of applications that are dependent on precise time measurements and increasing the resynchronization rate with an external global clock source, which counteracts the attainment of a desirable energy efficiency.

As an alternative to the system clock, we proposed RVEC, an original virtual clock with the property that the time count is strictly increasing and precise. Moreover, we used RVEC to build a High-Precision Global Clock (HPGC) solution which was free from resynchronization for a cluster of energy-efficient computers.

Our experimental evaluation of an implementation of RVEC in both the Linux system and the OpenVZ virtualization system as well as HPGC in Linux using a reference cluster of three energy-efficient computer systems showed that RVEC had negligible overhead and was highly precise. In addition, our results indicated that the HPGC synchronization solution is highly-scalable as it can synchronize up to 1,000 nodes/s and also that it could be performed offline, which could benefit energy-limited embedded systems as well. These preliminary results suggest that RVEC and HPGC can be an effective system clock and an effective global clock, respectively, especially for energy-efficient computer systems. However, more extensive evaluations are necessary. In particular, we intend to evaluate the impact of using RVEC and HPGC on large clusters of energy-efficient computer systems for distributed and parallel computing.

References

[1] Broomhead, T., Cremean, L., Ridoux, J., and Veitch, D. Virtualize everything but time. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[2] Cristian, F. A probabilistic approach to distributed clock synchronization. In *Distributed Computing Systems, 1989., 9th International Conference on* (jun 1989), pp. 288 –296.

[3] de Souza, A. F., de Souza, S. F., de Amorim, C. L., Lima, P., and Rounce, P. Hardware supported synchronization primitives for clusters. In *PDPTA'08* (2008), pp. 520–526.

[4] Intel. *IA-PC HPET (High Precision Event Timers) Specification*, Oct.. 2004.

[5] Jones, T., and Koenig, G. A. A clock synchronization strategy for minimizing clock variance at runtime in high-end computing environments. In *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing* (Washington, DC, USA, 2010), SBAC-PAD '10, IEEE Computer Society, pp. 207–214.

[6] Jones, T., Tauferner, A., and Inglett, T. Linux os jitter measurements at large node counts using a bluegene/l. Tech. rep., Nov. 2009.

[7] Meisner, D., Gold, B. T., and Wenisch, T. F. The powernap server architecture. *ACM Trans. Comput. Syst. 29*, 1 (Feb. 2011), 3:1–3:24.

[8] Mills, D. L. Network time protocol (version 3) specification, implementation and analysis. Tech. rep., 1992. http://www.faqs.org/rfcs/rfc1305.html.

[9] Muir, S. The seven deadly sins of distributed systems. In *First Workshop on Real, Large Distributed Systems, WORLDS'04* (Dec. 2004).

[10] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 361–376.

[11] Ridoux, J., and Veitch, D. Principles of robust timing over the internet. *Commun. ACM 53* (May 2010), 54–61.

[12] Stricker, T., and Gross, T. Optimizing memory system performance for communication in parallel computers. *SIGARCH Comput. Archit. News 23*, 2 (1995), 308–319.

[13] Tian, G.-S., Tian, Y.-C., and Fidge, C. High-precision relative clock synchronization using time stamp counters. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 69–78.

[14] Veitch, D., Ridoux, J., and Korada, S. B. Robust synchronization of absolute and difference clocks over networks. *IEEE/ACM Trans. Netw. 17* (April 2009), 417–430.

Notes

[1]A system clock with the SIP property assures that two consecutive clock readings, T_1 and T_2, will return time measurements $T_2 > T_1$ for any time interval between the two readings

[2]This hazard occurs due to the Linux migration procedure that is executed on task scheduling

[3]http://www.compasso.ufrj.br