16. Robson A E. Motion of a Short Arc in a Magnetic Field. Phys. Rev, 1956; 104: 15-16.
17. Coll B F, Sanders D M. Design of vacuum arc-based sources. Surf. Coat. Technol, 1996; 81: 42-51.
18. Miyano R, Saito T, Takikawa H, et al. Influence of gap length and pressure on medium vacuum arc with Ti cathode in various ambient gases. Thin Solid Films, 2002; 407: 221-226.
19. Swift P D. Macroparticles in films deposited by steered cathodic arc. J. Phys. D: Appl. Phys, 1996; 29:2025-2031.
20. Lang W C, Wang X H, Li M X. Multi-mode dynamic coupling alternating magnetic field on arc ion plating deposition arc source device, Authorization patent : ZL201210431983.5.
21. D.Meeker.Finite Element Method Magnetics. Available from: http://femm.foster-miller.net.
22. W C Lang, J Q Xiao, J Gong, et al. Influence of axisymmetric magnetic field on cathode spots movement in arc ion plating. Acta Metallurgica Sinica, 2010; 46(3): 372-379.

Multi-Granular Application Management Platform and Multi-Core-Aware Parallel Scheduling Model

Wei-Hua Bai[1,2], Jian-Qing Xi[1] *, Shao-Wei Huang[2], Jia-Xian Zhu[2]

1. School of Computer Science and Engineering, South China University of Technology, GuangZhou 510006, China
2. School of Computer Science, ZhaoQing University, ZhaoQing 526061, China

Abstract
This study proposes a novel multi-granular application management platform between PaaS and IaaS layers that uses application virtualization techniques. A multi-core-aware parallel scheduling model is then investigated on the platform. Assisted by fine-grained application units, new functions can be created by combining the application units in different granularities based on business requirements. The multi-core-aware parallel scheduling model was developed to process application requests, and not only increases system flexibility and applicability but also improves fine-grained computing resource allocation, resource utilization of the fundamental infrastructure, and system throughput.
Keywords: APPLICATION SERVICE, WEB SERVICE, MULTI-CORE-AWARE, PARALLEL SCHEDULING, MULTI-GRANULAR APPLICATION

1. Introduction

Alongside rapid design and development of applications in various computing fields, big data applications and the requirements of application services have grown increasingly complex and diverse. More IT resources, both hardware and software, are deployed in enterprise infrastructures than ever before. At present, PaaS platforms provide users with fundamental software research and development services, and SaaS is utilized to distribute the software, tailored in order to satisfy dynamic and mutable business requirements [1, 2].

Although PaaS applications can be used to create enterprise applications in SOA architectures [3], there are drawbacks in building business systems on PaaS platforms [4, 5]. For one, PaaS platforms provide coarse-grained modules, which only support particular types of business systems such as Customer Relationship Management (CRM), Office Automation (OA), Human Resources (HR), Supply Chain Management (SCM), and specific enterprise management software that maintains purchases, sales, and inventory [5]. Basically, PaaS platforms are unable to integrate diverse types of business into a comprehensive enterprise management system. Another major drawback is that PaaS platforms offer server platforms for fundamental or systematic development of business environments (services) to individual users [4, 6], but any pre-existing IT infrastructures cannot be fully protected or utilized [7]. Additionally, task schedulers in PaaS platforms are unaware of the fundamental infrastructures, (especially multi-cpu multi-core architectures,) and cannot fully utilize CPU resources [7, 8]. In heterogeneous, multi-CPU multi-core cloud computing environments, there yet remains an urgent need to successfully design a multi-core-aware parallel scheduler for application scheduling and execution based on the characteristics of applications and CPUs. The parallel execution of applications can be implemented by multi-thread techniques to increase the CPU utilization.

This study addressed the issues above from the following two primary aspects:

(1) We created a multi-granular application service layer between PaaS and IaaS using application virtualization techniques. The intermediate layer contains an application management platform, and the core module of the platform is a service management agent [9, 10]. Existing business logic is converted into web services in the SOA infrastructure in order to allow integration into new business systems. Developers can generate new functions by combining applications in different granularities based on the business logic.

(2) We also studied and designed a multi-core-aware parallel scheduling module on the application management platform. Based on application execution feedback and the known characteristics of multi-core CPUs, the scheduling module assigns applications to computing nodes comprehensively, in order to increase CPU utilization and improve the collaboration between computing nodes, CPUs on a particular node, and CPU cores.

2. Multi-granular Application Management Platform

2.1. The Structure of the Platform

Because enterprises may accumulate a large amount of resources over the course of their development, they cannot discard all existing IT infrastructures (including software such as system environments, business systems, and databases, as well as hardware such as servers, storage, and networks,) as they add new infrastructures [11]. In PaaS, users utilize applications and services use the computing unit with the API provided by IaaS. The IaaS manages the physical resources by virtualization technology, and provides an abstract computing unit to users and PaaS. The single layer structure of PaaS or IaaS does not meet the multivariate needs of user applications or environment; it should consider adding a virtualization layer between the Paas and IaaS, which can use both PaaS and IaaS.

In order to close the gap between IaaS and PaaS, combining both IaaS and Paas within a framework has been considered in much scholarship. In educational environments, Vaquero [12] presents EduCloud, which considers both IaaS and PaaS as supporting techologies for education. JcloudScale [13] is a Java-based middleware that supports building elastic applications on top of a public or private IaaS cloud. CloudMF [14] aims to add support for management of applications that may rely simultaneously on both IaaS and PaaS solutions. Calin and Dana [15] describe a hybrid architecture and implementation of a framework, which uses both the IaaS and PaaS to support setting up applications. In practical, cloud computing providers need a novel multi-granular application management platform to provide, generate and schedule applications based on PaaS and IaaS supporting.

As businesses expand, employing a multi-granular application management platform in enterprise cloud computing infrastructures allows them to satisfy new requirements by integrating existing IT infrastruc-

tures while quickly developing novel business functions.

The structure of the application management platform is displayed in Fig. 1. The platform integrates data in original enterprise systems into new systems through web services, and implements the business logic by combining the web services in SOA mode. The platform is encapsulated as independent business applications, and exposes development interfaces to third parties or business developers. All data is stored in the enterprise's IT infrastructures. In order to supply high performance for infrastructure users, all applications are scheduled by a parallel scheduling engine in the platform.

Users can manipulate the AaaS platform to build, deploy, and migrate applications, where the AaaS platform responds to user requests by managing the workflow queue and allocating computing resources with the multi-core-aware parallel scheduling engine. Because the application management platform has characteristics of both PaaS and IaaS layers, the platform can provide users with a more favorable software environment and application services by combining PaaS and IaaS without altering the cloud.

2.2. Application Models

On the AaaS platform, all applications are defined as templates that can be generated by combining finer-grained applications. There are two main advantages to this design: first, that fine-grained applications can make the system more flexible toward business requirements; and second, that finer-grained applications require less computing resources, which improves the effectiveness of computing resource allocation and fine-grained application parallelism at thread level. Because each fine-grained application is executed in a thread, and the business logic is created by combining the fine-grained application units through the programming interfaces, developers are allowed to convert their requirements to parallel applications on the platform in an easier manner.

Two types of applications are provided by the system: standard applications, including database operations (database connection pool management, queries, updates, insertions, and modifications,) file operations, inputs/outputs, and software interfaces; and customized applications added by users based on their development patterns and specifications. All applications in the AaaS platform can be described by the following model.

Definition 1: Application Units (*Apps*): the smallest units of applications in all types and granularities, i.e. the finest-grained applications. An application unit is an executable application with independent inputs and outputs on an AaaS platform. A *Apps* can be abstracted by a quintuple:

Apps ={App_Info, App_DInfo, App_Input, App_Output, App_DSP}

where *App_Info* represents the basic information of the application, including the application ID (*AppID*), name (*AppN*), the function description (*AppFDs*), and URL of the application package.

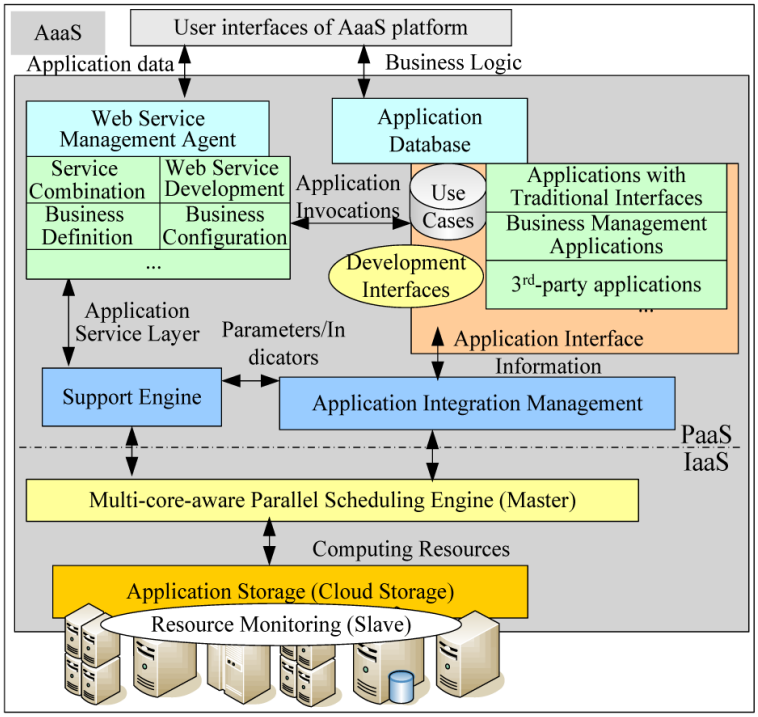

Figure 1. Structure of multi-granular application management platform

App_Info={ AppID, AppN, AppFDs, URL }, let *AppN* and *AppFDs* be two subsets of internal keywords.

App_DInfo describes the additional information of the application, including the name of the application type/field (*AppTN*), application provider ID (*AppPID*), access levels (*Authority*); then *App_DInfo={ InputNu, InputDN, InputDT }*.

App_Input indicates the input parameters of the application – the number of input parameters (*InputNu*), parameter names (*InputDN*), and input parameter types (*InputDT, InputDT∈ Setof(DataType)*) – then *App_Input={ InputNu, InputDN, InputDT }*.

App_Output denotes the outputs of the application, including the names of the outputs (*OutputDN*) and output data types (*OutputDT, OutputDT∈ Setof(DataType)*); then *App_Output={ OutputDN, OutputDT }*.

App_DSP describes the characteristics of executing the application on CPU cores. For a granularity flag (*Gflag)*, if *Gflag=T*, then the application is a fine-grained application with independent functions; if *Gflag=F*, then the application is a combined application. A type of calculation (*CalType)*, is valid if *Gflag=T*.

Definition 2: Application Flow (AppF) is an application group comprised of n (n≥1) application units (Apps) or application flows (AppF) in a particular order. By definition, an application flow is allowed to recursively combine application flows. An application flow can be represented by a tuple:

AppF = <Vapp, Era>

where Vapp={ $Vapp_i$ | $Vapp_i \in$ Setof(Apps, AppF), 1≤i≤n or i=b or i=e } denotes an independent application set (called a "set of vertices".) Vappb and Vappe are two virtual applications, which represent the beginning and the end points in the application flow. Era={<$Vapp_i$,$Vapp_j$> | $Vapp_i$, $Vapp_j \in$ Setof(Apps, AppF), 1≤i≤n or i=b, 1≤j≤n or j=e, i≠j } describes a pair of applications, indicating the relation between the caller and callee.

The granularities of application flows are determined by the amount of resources occupied by the application flows and their occupation time. In general, the more application units are in an application flow, the more computing resources the application flow will use and the longer their occupation time is.

Definition 3: The granularities of application flows can be classified into three categories according to the complexity of the application flows. Fine-grained application flows (*SG_AppF*), medium-grained application flows (*MG_AppF*), and coarse-grained application flows (*BG_AppF*) are respectively defined as follows.

(1) Fine-grained application flows (SG_AppF) are those comprised of a single application unit (Apps), i.e. Vapp={ $Vapp_b$,$Vapp_1$, $Vapp_e$ }, Era={<$Vapp_b$, $Vapp_1$>, <$Vapp_1$,$Vapp_e$>}, as shown in Fig. 2(a).

(2) Medium-grained application flows (MG_AppF) are comprised of n (n>1) application units (AppF), i.e. Vapp={ $Vapp_b$, $Vapp_1$, $Vapp_2$, $Vapp_n$,$Vapp_e$ }, Era={<$Vapp_b$, $Vapp_i$>, <$Vapp_j$, $Vapp_e$>}, as shown in Fig. 2(b).

(3) Coarse-grained application flows (BG_AppF) are complex application flows that are generated by recursively combining application units (Apps) and application flows (AppF), as shown in Fig. 2(c). Because the application is designed as an execution package with a clear input dataset and an output throughout, its separate parts can be integrated.

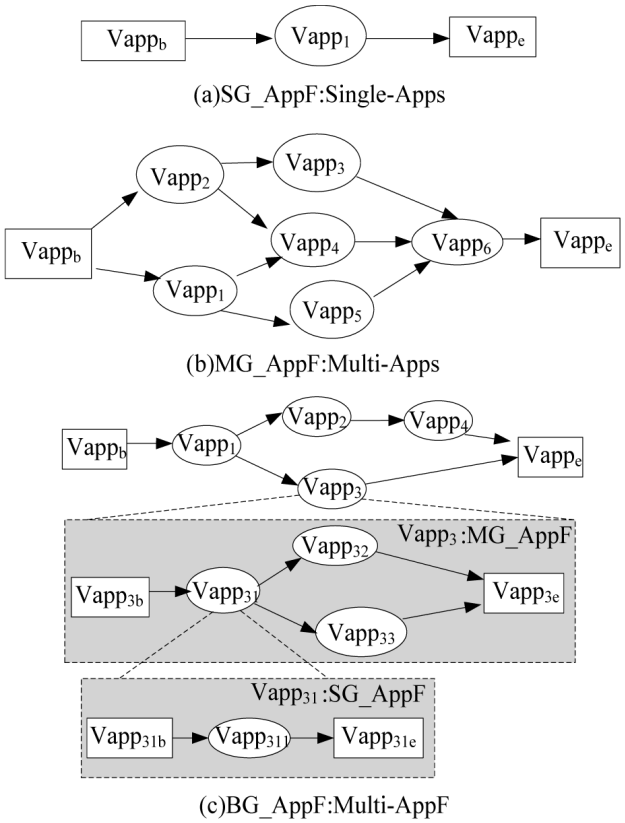

Figure 2. Multi-granular application flows

The above definitions infer that application flow classification presents the following advantages:

(1) The logic structures of application flows can be easily described by and stored in XML files. For example, shown in Fig.3, a distributed AppF that calculates super π can be defined by the AppF template below.

During the scheduling process, an execution sequence of the application flow which will be parallel-scheduled can be generated by applying a breadth-first traversal method to the XML files (see Section 3.3).

```
<AppF ID="super pi_test" type="BG_AppF">
<Vappb ID="super pi_test_begin" InputNu="0" InputDT="double" Values="1000000.0"
ChildID="Calculation-pi-1,Calculation-pi-2"></Vappb>
<Vapp ID="Calculation-pi-1" type="Apps" InputNu="1" InputVappID="super pi_test_begin"></Vapp>
<Vapp ID="Calculation-pi-2" type="Apps" InputNu="1" InputVappID="super pi_test_begin"></Vapp>
<Vapp ID="Calculation-pi-merge" type="Apps" InputNu="2"
InputVappID="Calculation-pi-1,Calculation-pi-2"></Vapp>
<Vapp ID="Output_file" type="AppF" InputNu="1" InputVappID="Calculation-pi-merge"></Vapp>
<Vappe ID="super pi_test_end" InputNu="1" InputDT="Output_file"></Vappe>
</AppF>
```

Figure 3. An example AppF of calculating super π

(2) In the AaaS platform, recursively defining AppF allows reuse of applications in all granularities.

(3) According to the Apps definition, it is easier to generate an AppF from two Apps ($Apps_i$ and $Apps_j$) by calculating the similarities of their inputs and outputs. It is also beneficial to calculate the similarities of the inputs and outputs of the AppF and business logic, e.g., *similarity (BLogic.InputVar, $Apps_i$.Input)* and *similarity (BLogic.OutputD, $Apps_j$.Output)*. When these two similarities are greater than the given similarity thresholds ζ and η, the AppF can serve as the application flow of the business logic in a system. (We can use the technology of web service composition for supporting these processes.)

The AaaS platform continues using or improves mature applications/software. A full life cycle, from infrastructure design, interface design, and project management to system execution support and monitoring, can be completed through the AaaS platform. Moreover, the AaaS platform has coarse-grained interface design applications, IT project management applications (such as progress control, quality management, testing management, defect management, configuration management, product release, personnel management, and search and code generation,) business system execution support, and monitoring applications. The AaaS platform also supports integration and collaboration with other business systems.

In order to reuse the Apps and AppF in the system, we define a set of rules for copying and sharing. E.g., UOsCr (User Objects Copy rules), UOsSr (User Objects Share rules), ORrSr(Object Response result Share rules), and so on. For example, UOsCr: If the owner $\alpha_i(1\le i\le m)$of the object O_k $(1\le k\le n)$ grant the copy right to user $\alpha_j(1\le j\le m)$, then α_j can copy O_k to the α_j home. UOsCr can be formulated as:

$$\forall O_K \in \{Apps | AppF, Apps, AppF \in Object\}$$
$$1\le k\le n$$

O_k Belong toα_i

$$a_i \frac{Copy(O_K)}{grant} \Rightarrow a_j$$

UOsCr: Home(α_j) accept O_k as $O_k^{'}$

This issue is not further discussed because it is out of the scope of this study.

2.3. Implementation and Management of Application Virtualization

The process purpose of web service virtualization is to build a bridge between business logic and system implementation. Business logic can be completed by mapping it to one or more web services. The Web Service Management Agent (WSMA) [9, 10] is responsible for web service registration and management, web service matching and combination configuration, and output serialization.

As shown in Fig. 4, the application virtualization system consists of a computing resource layer (IT infrastructures), an application invocation layer, an application conversion layer, and an application combination layer.

(1) The computing resource layer is constructed using enterprise IT infrastructures. In order to supply customer services, all execution packages and existing web services are distributed across computing nodes (servers). Applications in all granularities can be deployed and executed on these computing nodes.

(2) The application invocation layer records the run-time statuses (busy or idle) of the computing nodes and dynamically invokes applications (web services, Apps) on appropriate nodes by demand through WSMA and the parallel scheduling engine.

(3) The application conversion layer converts the business applications into web services with application templates and provides them to users.

(4) The application combination layer creates new business systems based on users' coarse-grained applications obtained according to business requirements.

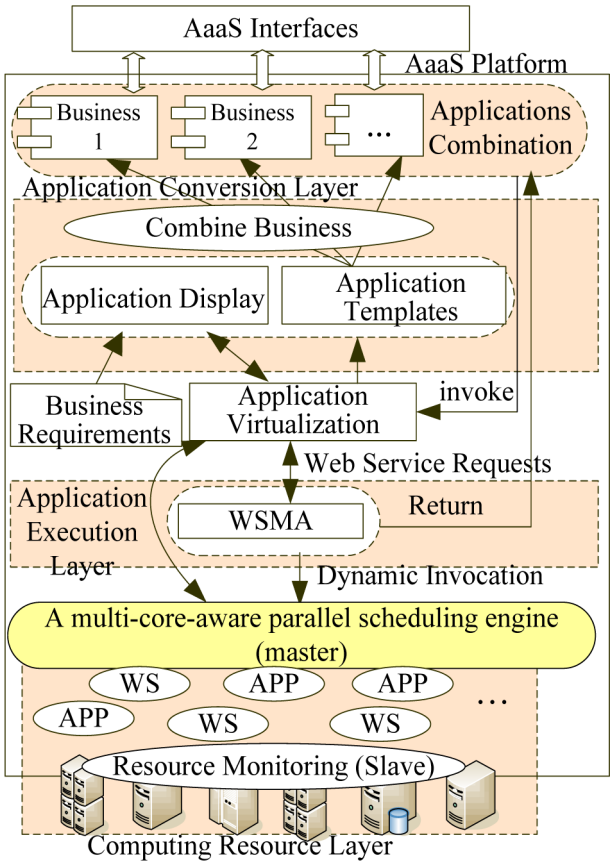

Figure 4. Implementation and management of application virtualization

Users can develop their business systems through AaaS programming interfaces. The interfaces support the combination of applications in all granularities for business development driven by business requirements. The interfaces also display applications in all granularities to users and integrate existing business data into a new system under relevant standards and specifications using web service templates.

In the AaaS platform, application scheduling and execution is supported by a multi-core-aware parallel scheduling model, which serves as the executor on the platform.

3. Multi-Core-Aware Parallel Scheduling Model

In cloud infrastructures, there exists an overarching trend where computing nodes are equipped with multi-chip multiprocessor architectures. To fully utilize the computing resources and improve the performance of the application management platform, we propose a multi-core-aware parallel scheduling engine. (This paper only presents the logic structure of the parallel scheduling model and related experimental results due to space limitations.)

3.1. Structure of Parallel Scheduling Model

On the AaaS platform, we designed a two-level multi-core-aware parallel scheduling model that includes a workflow scheduler in the master node and a task scheduler in each computing node.

The structure of the multi-core-aware parallel scheduling model is displayed in Fig. 5.

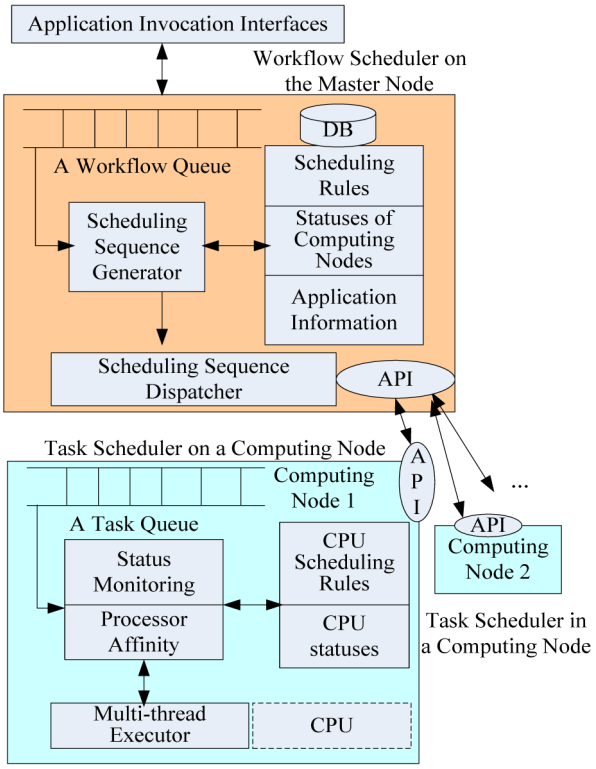

Figure 5. Structure of multi-core-aware parallel scheduling

The workflow scheduler in the master node inserts application invocation requests into the workflow queue, then generates scheduling configuration XML files through the scheduling sequence generator (XML Code Generator) based on the pre-defined "scheduling rules", "statuses of computing nodes", and application information (including the information of last execution, the application type, and the execution time.) Finally, the workflow scheduler assigns the applications to corresponding computing nodes through the scheduling sequence dispatcher and sends responses to users.

Users can specify scheduling rules that control the schedule of application invocation requests. If no scheduling rule is set, a First-Come-First-Serve (FCFS) rule, which assigns applications to idle computing nodes to maintain workload balance among the nodes, is applied by default.

The statuses of computing nodes include the architecture information of CPUs and the utilization of computing resources (CPU, memory, storage, and network) on the node. This information is collected by status monitors on computing nodes and reported to the master node through heartbeat messages. Said

heartbeat messages also indicate whether the computing nodes are in active status.

The application information module maintains the performance characteristics of applications. Based on this information, applications are classified into categories and assigned to computing nodes, the characteristics of which are similar to the performance characteristics of the applications, in order to increase the system throughput.

The task schedulers on computer nodes receive scheduling configuration XML files and application packages (if the applications are not deployed on the computing nodes) from the scheduling sequence dispatcher on the master node, and then insert the tasks into task queues. (Again, the default FCFS scheduling rule is applied if no customized rule is specified.) Then, the corresponding threads of the applications are bounded to particular CPU cores with Processor Affinity techniques in order to ensure favorable performance.

The status monitor records the thread statuses of applications on CPUs, which include CPU cycles, cache miss rate, the execution time of the thread, and the number of instructions completed. With the help of this information, our module obtains application execution characteristics and changes the scheduling rules (policies) based on the feedback of the application execution if necessary.

The multi-thread executor uses Processor Affinity techniques, which bounds threads to specified CPU cores and assigns threads to idle CPUs or CPU cores on alternative chips. The executor is able to increase CPU utilization by reducing the amount of application migration among CPU cores, the waiting time, and the cache miss rate.

3.2. Multi-Core-Aware Feedback Model

It is commonly known that thread level parallelism is a critical issue in multi-chip and multi-core environments. The multi-core-aware feedback model is a crucial component of the AaaS platform. We designed a feedback mechanism [16, 17], in which the execution statuses of application units (Apps, see ***Definition 1***) in CPU cores collected by status monitors on computing nodes are sent back to the multi-core-aware parallel scheduling engine to ensure optimized scheduling policies, appropriately balancing the workloads of computing nodes and CPU cores and improving overall system performance/throughput.

To effectively monitor the running of a Apps in a core in a computing node, using Processor Affinity techniques and the API of the Linux kernel, we built our design with two main classes, *ThreadController* and *AppsMonitor*. All the Apps are stared by the *ThreadController* class and the thread PID of the Apps is registered to the *AppsMonitor* class to obtain execution quota information. A sequence diagram of the process is shown in Fig. 6.

The monitor of a computing node consists of the *ThreadController* and *AppsMonitor*. The invoker informs the *ThreadController* to create a thread for the Apps. The *ThreadController* uses *AppsRunner(Url)* to star the Apps, and sends the *Pid* of the thread to the *AppsMontior*. After registering the *Pid* of the thread, *AppsMonitor* obtains the running information in the core – this can be done at the runtime of the Apps. All the *Apps*' running information (***Definition 4***) can be used by the multi-core-aware feedback model.

Definition 4: The execution properties of an application unit (Apps) in each thread are defined as ***Tp={t_c,ins,cyc,p_f,c_m,cr,total_t}***, where

t_c, the task-clock, denotes the execution time of the thread on a particular processor. It is a static property and can be collected when the thread is completed.

ins, the instructions, indicates the number of instructions completed after the thread began. It is a dynamic property and can be collected while the thread is executing.

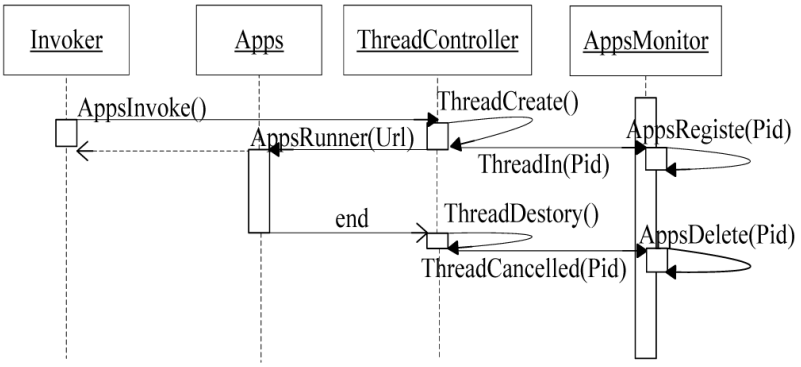

Figure 6. Sequence diagram of Apps monitor in computing node

cyc, the cycles, represents the number of CPU cycles consumed by the thread. It is a dynamic property, similar to *ins*.

p_f, page-faults, describes the number of page faults that have occurred in the kernel. It is also a dynamic property, like *ins* and *cyc*.

c_m, cache-misses, denotes the number of cache misses. It is also a dynamic property.

total_t, the total-time, indicates the execution time of the thread. It is a static property, similar to *t_c*.

Since computing nodes may be equipped with alternative CPU architectures, in which the frequencies of CPU cores, cache size (L1 and L2), memory size, memory frequency, and external storage I/O performance may vary considerably, the execution properties ***Tp*** of application units (Apps) and the feedback of the execution on CPUs may also vary quite widely.

Definition 5: The performance metrics of an application unit in each thread are defined as ***Te={RTOC,IPC,PFMPI,CMKPI,CRKPI}***, where

RTOC=*(t_c/total_t)*100%*, indicates the percentage of CPU usage occupied by the thread. *RTOC* is used to classify the thread type – if *RTOC* is close to 1, the thread is CPU-intensive; otherwise, it is I/O-intensive.

IPC=*ins/cyc*, instructions per cycle, denotes the number of instructions completed in a CPU cycle, which measures the degree of CPU features being used by the thread. The greater the *IPC* is, the more CPU features are used.

PFMPI=*p_f*10⁶/ins*, counts the number of page faults in each 10^6 instructions. *PFMPI* measures the dependency of the thread on the interactions between internal and external storage systems. The greater the *PFMPI* is, the greater the number of page faults during execution and the higher the dependency.

CMKPI=*c_m*10³/ins*, counts the number of cache misses in each 10^3 instructions, i.e., *c_m*10³/ins*,. *CMKPI* measures the cache usage information of the CPU core. Because multiple cores share a cache other threads on other CPU cores interfere with the cache usage of a given CPU core.

CRKPI=*c_r*10³/ins*, counts the number of cache hits in each 10^3 instructions. *CRKPI* is measured in the same way as *CMKPI*.

CMKPI and *CRKPI* are two metrics used for dynamic optimization of application execution. For example, to make multiple threads share a cache, our module inserts threads with similar *CMKPI* to the task queue of one CPU core, or avoids threads with high *CRKPI* in the task queues of CPU cores that share cache.

We established an application unit testing database for classifying the types of application units Apps (such as CPU-intensive applications, I/O-intensive applications, memory-dependent applications, or CPU-intensive and memory dependent applications.) We selected an application in each type as a representative and used the performance metrics *Te* of each, calculated from the feedback of its execution on computing nodes, as the benchmark of its type. The benchmark of the performance metrics is denoted by ***BTe***.

We assumed there were *m* types of application units in a system, and that the benchmark of performance metrics in the *k*th application type is denoted by $\boldsymbol{BTe_k}$=**{**$\boldsymbol{RTOC_k}$, $\boldsymbol{IPC_k}$, $\boldsymbol{PFMPI_k}$, $\boldsymbol{CMKPI_k}$, $\boldsymbol{CRKPI_k}$**}**, marked by $\{r_k, i_k, p_k, cm_k, cr_k\}$. Given an application type *t*, and its performance metrics $\boldsymbol{Te_t}$=**{**$\boldsymbol{RTOC_t}$, $\boldsymbol{IPC_t}$, $\boldsymbol{PFMPI_t}$, $\boldsymbol{CMKPI_t}$, $\boldsymbol{CRKPI_t}$**}**, marked by $\{r_t, i_k, p_t, cm_t, cr_t\}$.

Eq. (1) can be used to calculate the Euclidean distance between Te_t and BTe_k as the distance of the two application types:

$$D_t^k = |BTe_k - Te_k| = \sqrt{(r_k - r_t)^2 + (i_k - i_t)^2 + (p_k - p_t)^2 + (cm_k - cm_t)^2 + (cr_k - cr_t)^2} \quad k = 1, 2, ..., m \tag{1}$$

The type of an application unit (Apps) can be determined using Rule *AppsTr* Eq. (2) :

$$AppsTr: if\,(D_t^l - \min_{k=1,2,...m}(D_t^k)) = 0 \Rightarrow t\arg et_{type} = l \tag{2}$$

In each computing node on the AaaS platform, performance monitors collect the execution properties *Tp* of each thread in application units (Apps) through the following two methods:

(1) The static method (Two-Checkpoint method), which sets a checkpoint at the start point and end point of the execution on CPU cores. This method estimates the overall performance of applications, and is an important parameter of workflow scheduling decision-making with the parallel scheduler on the master node.

The static method is employed to estimate and analyze the performance of an application with an analyzer after the application has been deployed in the platform and before being invoked. To this effect, the overhead of the static method is negligible.

(2) The dynamic method, which dynamically sets checkpoints during the execution on CPU cores. The times of the checkpoints can be determined using Random Checkpoint methods or Interval Checkpoint methods. The dynamic method provides indicators that monitor performance metrics trends and creates thread scheduling prediction models on computing nodes.

Due to the potentially large number of threads in a system, applying the dynamic method to monitor all threads and collect their data introduces unacceptably high overhead to the system, reducing overall system performance. Thus, the dynamic method is only used after imbalance in workloads on CPU cores of a computing node is identified.

3.3. Parallel Scheduling Strategies

Users submit application requests in application flows (AppF). By definition, application units (Apps) are the basic units of AppF. Because modern applications are designed to be independent, with clear inputs and data outputs, each Apps can be considered an independent application service with a form of execution package in the system. Thus, Apps is the finest unit scheduled and executed in parallel on the AaaS platform. The AaaS platform exists to complete user requests – only partially completing said requests renders the platform useless. The essence of the parallel scheduling policies in an AaaS platform is to fully and successfully process user application requests.

Based on the feedback *Tp* (see ***Definition 4***) and *Te* (see ***Definition 5***) of each Apps running on CPU cores, the computing resources required by the Apps can be estimated in advance. Because Apps are small applications that only require limited computing resources, we applied the following scheduling policy in the parallel scheduler: FCFS and backfilling strategy [18, 19] are used to manage AppF queues, in which computing resources are pre-allocated to each Apps in an AppF. A scheduling decision is made based on all Apps in the AppF. (Only the management and the scheduling process of the AppF request queue is discussed in this paper.) The scheduling process is depicted in Fig. 7.

As shown in Fig. 7, to implement the management and scheduling of the AppF request queue as well as monitor the workload of computing nodes, the following three structures were established:

(1) $AppFWorkQueue=\{AppF_1, AppF_2, \ldots AppF_n\}$, $(AppF_i \in XML\ file)$, a queue of AppF requests submitted by users. Each element in the queue is an XML file of the corresponding AppF.

(2) $AppsList=\{\{Vapp_i, \{WaitCount, AppsObject\}\}, \ldots\}$, *a Hashmap<K key, V value>*, in which each element abstracts an Apps, where the *key* is $Vapp_i$(Apps ID) , and the *value* is a data object, including the number of predecessors *WaitCount* and the basic information of Apps *AppsObject*.

(3) *NodeList={{CNodeID,Node_ActionApps_ArrayList},...}*, an *ArrayList*, in which each element represents the workload of a computing node, i.e. the currently active Apps; the data consists of the node id *CNodeID* and a sequence of corresponding Apps *Node_ActionApps_ArrayList*.

The pseudo-codes of the AppF scheduling in the function Schedule_AppF() is displayed below.

```
function Schedule_AppF( )
    xmlfile activeAppF;  //an AppF is saved as a xml file;
     DSaovType daaov;  //a defined data type DSaovType saved an AppF graph
     ArrayList tpol;    //a Topological Order list of an AppF graph
     CNodeId cnid; //a node's ID while((AppFWorkQueue.poll( )→activeAppF)==True)
    //①get the top element from AppFWorkQueue
        BFS(activeAppF)→daaov;  //②use the Breadth First Search Algorithm to create the AppF //graph and saves
it to daaov
       Topo_Order(daaov)→tpol;  // ②create the Topological Order list
       if(Preallocated(tpol))      //③if successes to pre-allocated the resource for each Apps in the tpol
        for each(Appsi in tpol)
           Object V=new Object(WaitCount, AppsObject); //WaitCount is the number of the //parents of the Appsi
            AppsList.put(Vappi,V);   //④add the element to the AppsList
    ScheduleXmlGenerate(V,AppFSchedulexml); //⑤add the Appsi to a scheduling xml file
          InsertApps(cnid, V);  //⑥insert the Appsi to a node which will execute the Appsi
        end for
        Exc_Schedule(AppFSchedulexml);  //⑦use the AppFSchedulexml to execute the Apps
      else    ...   // no enough resources to execute the AppF, do something else
      end if
      AppFWorkQueue.remove( );  //remove the element from AppFWorkQueue
    end while;
```

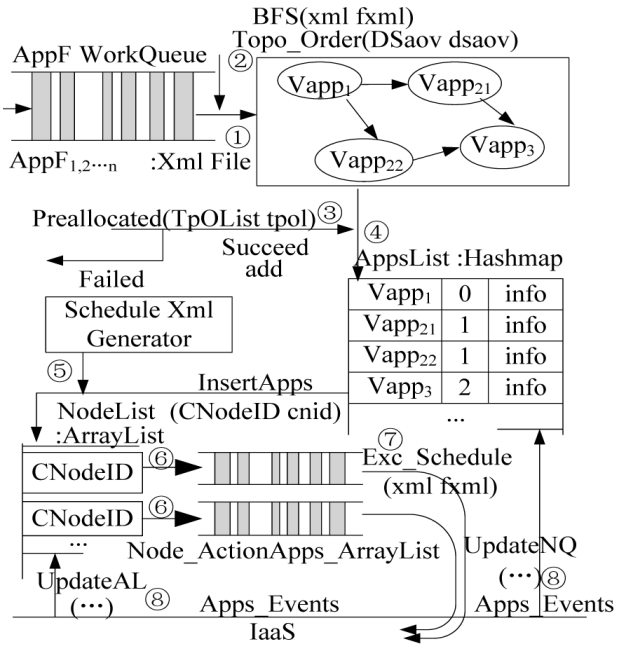

Figure 7. Management of AppF request queue

As shown in Fig. 7, when an Apps is completed on a computing node, ⑧ triggers Apps events to update/remove properties of the corresponding Apps by *AppsID* through *UpdateAL()* and *UpdateNQ()*; in other words, removing the Apps from AppsList, then updating the property *WaitCount* of its successors ((WaitCount-1)→WaitCount) and finally removing the workload information from *Node_ActionApps_ArrayList*.

The AaaS platform uses a master/slave two-level parallel scheduling model. The parallel scheduling policies can be divided into the following layers:

(1) Workflow scheduling policies, in which coarse-grained scheduling policies are conducted by the workflow scheduler on the master node. The workflow scheduler mainly works on application requests submitted by users based on classic scheduling algorithms, such as FCFS, multi-level feedback queue scheduling algorithms, the shortest-job-first scheduling algorithm, and priority-based scheduling algorithms.

(2) Application-type-based scheduling policies, in which applications on the AaaS platform can be invoked multiple times by users. Based on characteristic information given by feedback of the execution on computing nodes, applications can be classified as I/O-intensive applications, (such as file accesses, building indices of files, and receiving user inputs,) or CPU-intensive applications, such as algorithm computation. The application-type-based scheduling policies consider the application types and the performance of computing nodes in the scheduling decision-making. For example, I/O-intensive applications will be assigned to low-performance computing nodes; while CPU-intensive applications will be executed on high-performance computing nodes. This type of scheduling methods can fully utilize the CPU features on the computing nodes.

(3) Task scheduling policies, in which the task scheduler applies a multi-level feedback queue algorithm on computing nodes to assign task scheduling based on the performance of the computing node. The performance of the computing node is determined primarily by CPU performance. Experimental results [16] have shown that there are a maximum number of threads supported by any multi-core CPU architecture. If the number of threads is greater than the maximum value, the overall performance of the computing node will decrease as the number of threads increases.

(4) Multi-thread scheduling policies, in which, based on the feedback, threads in the same type are assigned to one CPU core in the multi-processor multi-core architecture of the computing node. For example, I/O-intensive multi-threads or threads with similar cache miss rates can be bounded to one CPU core in order to increase CPU utilization.

On the AaaS platform, the two-level parallel scheduling model optimizes scheduling rules (policies) based on feedback regarding application execution on computing nodes. The primary advantage of the platform is that it can maximally increase CPU performance and system throughput by applying a feedback mechanism to the multi-core-aware parallel scheduling model, optimizing application execution.

3.4. WETOBAND-An AaaS Platform

We design and implement *WETOBAND* as an AaaS platform for testing. Some UI of the *WETOBAND* is shown in Fig. 8.

4. Experiments and Discussion

4.1. Test Bed

Hardware: 2×Dell PowerEdge T720 (2×CPUs: Xeon 6-core E5-2630 2.3G, 12 cores in total); 1×Inspur NF8560M2 (4×CPU: Intel Xeon 6-core E7-4807 1.86G, 24 cores in total); 3×Lenovo M4336 (1×CPU: Intel 4-Core i7-3770 3.4G), 12 cores in total.

Software: OS-Linux kernel 2.6.21; AaaS Platform-JAVA1.6, JAVA RMI, JAVA Jna; Web-Tomcat 6.5, Apache.

The objectives of experiments were to verify the feasibility of the multi-granular application on the AaaS platform, the feasibility and effectiveness of the multi-core-aware parallel scheduling model, and the feasibility and effectiveness of the feedback mechanism during parallel scheduling optimization.

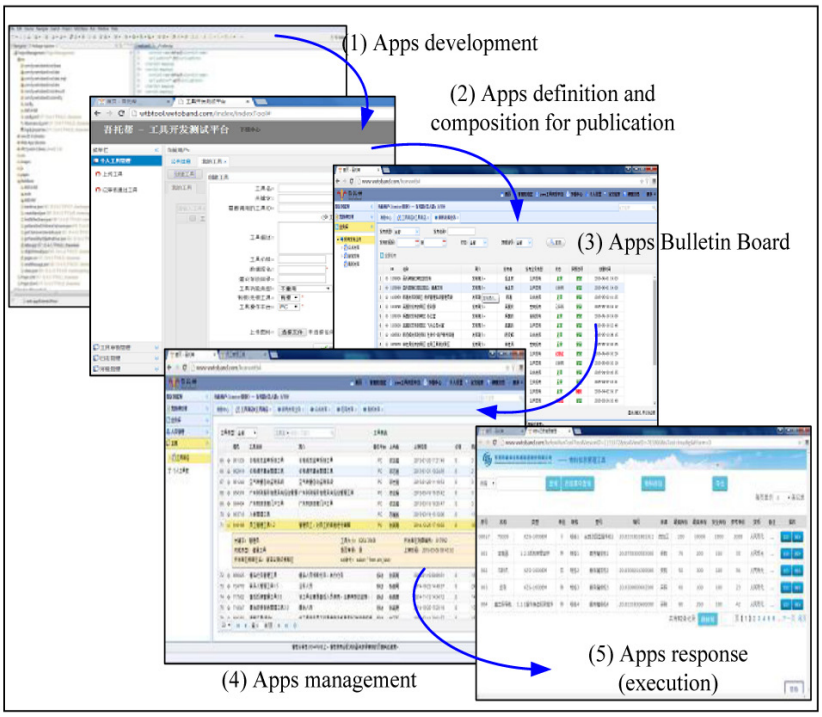

Figure 8. WETOBAND-A Multi-granular Application Management Platform

Test bed setting: one Lenovo M4336 was used as the master node and a computing node; other servers were configured as computing nodes. In our experiments, we used three clients to simulate application requests and submitted them to the master node. We selected a representative application for each type of application mentioned above. Each application that we selected had a clear input and controllable output information and computation scale. The application requests were as follows.

- $\pi = 4\sum_{k=0}^{n} \frac{(-1)^k}{2k+1}, n = 10^7$, an *SG_AppF* application with CPU-intensive *Apps* (***Definition 3*** and Rule *AppsTr-Eq.2*). The AppF type is called "A".
- *Super π (n=10⁹),* calculated by divided into *M* (*M ≤ the number of CPU cores in the system,* set to 6 in our experiments) tasks, in which *(M-1)* tasks calculate $(-1)^k/(2k+1)$, *(k=1,2,...m)* in parallel and the *M*th task calculates the sum of the intermediate results. This is an CPU-intensive application flow of *MG_AppF* type (***Definition 3*** and Rule *AppsTr-Eq.2*). The AppF type is called "B".
- *WordCount*, an application that counts the number of words in a file. In our experiments, we used a text file (file size 128 MBytes) and the output of *WordCount* was written to a specified file. According to Definition 3 and Rule 1, *WordCount* is an I/O-intensive application of *BG_AppF* type. The AppF type is called "C".
- The multiplication of two matrices of order *N (N=10³)* . Matrix multiplication is a CPU-intensive and memory dependent application of *MG_AppF* type (***Definition 3*** and Rule *AppsTr-Eq.2*). The AppF type is called "D".
- Calculation of *Kmeans (K=10)* of given *n (n=10⁴)* points in a two-dimensional space, where the result is written to an external file. Under Definition 3 and Rule 1, *Kmeans* algorithm is a CPU-intensive, I/O-intensive, and memory dependent application of *BG_AppF* type. The AppF type is called "E".

We did not apply any optimization to the five types of applications. To evaluate the system throughput, all applications must return «True» after completion.

We created three test datasets in the experiments based on the scale of the infrastructure and the scale of computation shown in Table 1.

Table 1. Test Dataset Information.

Name	Application Type-Number of AppF
TS-1	Total Size: 100; A-25; B-20; C-10; D-25; E-20
TS-2	Total Size: 200; A-50; B-40; C-20; D-40; E-50
TS-3	Total Size: 400; The number of *AppF* in each type is randomly generated; the number of *AppF* in type C is less than or equal to 30.

* «Total Size» represents the number of application requests from users, i.e., the number of AppF in the AppF-WorkQueue. The values that follow type names indicate the number of AppF in the types. The invocation sequence is randomly generated and recorded; the sequence will be used in the next experiment.

4.2. Analysis of Experimental Results

We completed the test cases in the experiments described in Table 1 (TS-1: 100 Apps requests in total; TS-2: 200 AppF requests generated; TS-3: 400 AppF requests randomly generated.) The execution time of all test cases was recorded. The default scheduling policy (applying balanced workload across computing nodes and FCFS) was applied as the benchmark case. We also evaluated execution time when the multi-core-aware parallel scheduling model (using balanced workload across computing nodes, FCFS, a threshold on the number of threads in each core, a threshold on the number of threads in each type on a computing node, and thread bounding) in the experiments. The execution information of the benchmark case was used as feedback in the feedback mechanism. The experimental results are displayed in Fig. 9-11.

During the experiments, we used the following six configurations: 1×M4336 (1 node, 4 cores); 2×M4336 (2 nodes, 8 cores); 3×M4336(3 nodes, 12 cores); 3×M4336 and 1×T720 (4 nodes, 24 cores); 3×M4336 and 2× T720 (5 nodes, 36 cores); and 3×M4336, 2× T720 and 1×NF8560M2 (6 nodes, 60 cores). As shown in Fig. 9-11, we made the following observations.

(1) The execution time of applications in Type C (I/O-intensive/memory dependent) was the greatest among all experimental applications, which is a major factor – testing took quite a long time.

(2) The policy of balancing workload across computing nodes and FCFS, as mentioned above, was applied in the benchmark cases. If using the default rule on TS-1 cases, 10 applications in Type C may be assigned to one computing node (depending on the order of requests,) which would significantly reduce system performance. The more tasks wait on the

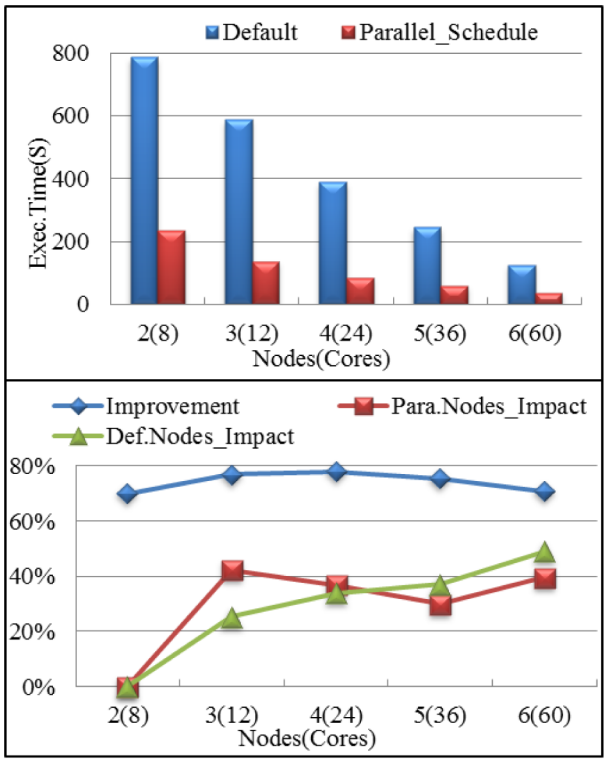

Figure 10. Experimental results-TS-2

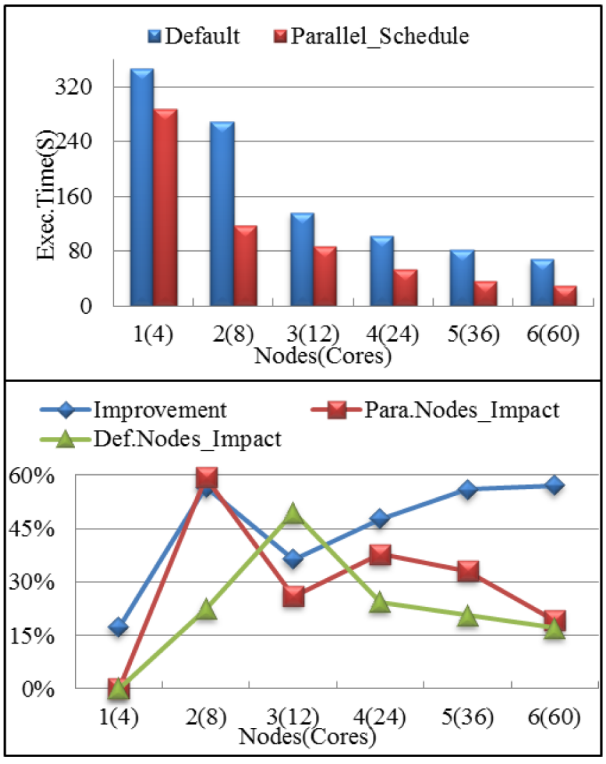

Figure 9. Experimental results-TS-1

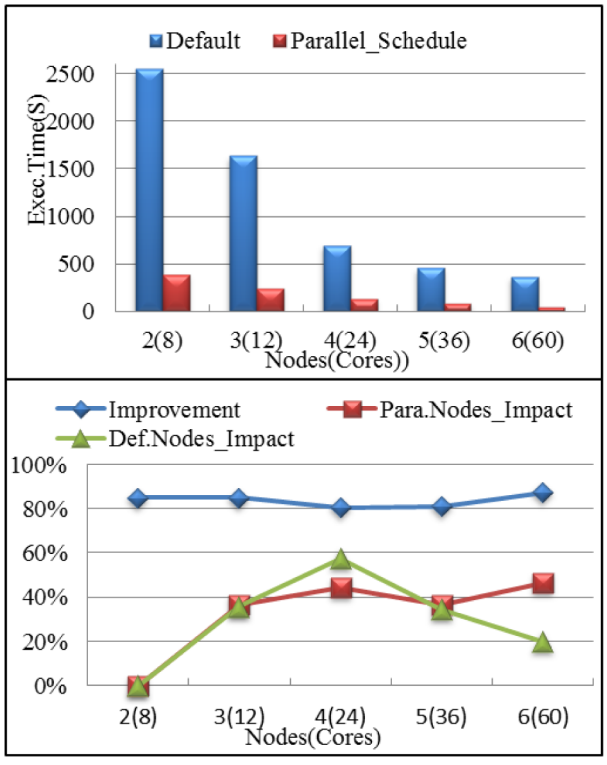

Figure 11. Experimental results-TS-3

node, the longer the execution of tasks will take. In the worst case, the entire system may crash; thusly, there were at least two computing nodes deployed in TS-2 and TS-3.

(3) Based on application feedback information and the rule under which thresholds were set on the number of threads in each core and on the number of threads in each type, our parallel scheduling model avoided performance degradation by assigning applications in the same type to multiple computing nodes. In addition to the 17.3% performance improvement shown in Fig. 9, our parallel scheduling model achieved more than 35% performance improvement in all other cases. The performance improvement became increasingly significant (over 80%) as the number of applications grew, as shown in Fig. 11. For example, 8 applications in Type C were waiting on a computing node, and the system took 536 seconds to complete these applications. If these applications were assigned to two computing nodes in a balanced manner, they could be completed in 47 seconds – only one eleventh of the execution time in the previous case.

(4) In the benchmark experiments, system performance was determined solely by the number of computing nodes. When the number of computing nodes was increased, system performance was enhanced accordingly. Although the balanced application distribution method always attempts to assign complex applications to different computing nodes, this method cannot effectively avoid cases where complex applications in the same types are running on one node. System performance cannot be improved sufficiently using this method, as it is dependent on the order of application requests.

(5) Based on the feedback mechanism, our parallel scheduling model combines thread-bounding techniques with the rule under which thresholds are set on the number of threads in each core and on the number of threads in each type, in order to balance the workload among CPU cores in a computing node and reduce the likelihood of system performance degradation by avoiding applications of the same type waiting on one node. The scheduling model can also reduce thread migration among CPU chips or cores in order to increase the cache hit rate. Therefore, this model improves overall system performance as the number of computing nodes and CPU cores increases. The more tasks running on the system, the greater performance improvement can be achieved using the scheduling method.

(6) A maximum value was set as a threshold to limit the number of threads on a node. The value of the threshold depended on the number of CPU cores on the node. Before reaching the threshold, the system performance linearly increased with the number of threads; but if the number of threads exceeded the threshold, the system performance remained essentially unchanged. Even when the number of threads hit another higher threshold, the computing node entered a «dead» state. There is a dead threshold on the number of applications of each type on a computing node, where the value of the dead threshold depends on the number of CPU cores, memory, and I/O. For example, the dead threshold of applications of Type C on an M4336 node is 9.

(7) The AaaS platform provides users with advanced parallel programming interfaces for multi-core systems using JAVA language, which allows users to effectively utilize CPU cores on computing nodes. By configuring or combing XML files, related applications can be collaboratively executed in parallel. Take the *Super* π ($n=10^9$) computation for example – based on the number of CPU cores (M) in the system, the computation was divided into (M-1) sub-tasks that calculated π (the size of input parameter was n/M of total inputs) and a task that combines the intermediate results. If the computation is completed on a single-node system, the computation acceleration is close to the number of CPU cores; the WordCount application can be processed in parallel similarly if the input file has been cut into multiple independent data blocks.

5. Conclusions

Within cloud computing environments, requirements for big data processing and application services have become more complex and diverse in recent years. Based on an analysis of collaboration and combination of PaaS and IaaS platforms under cloud computing infrastructures, this study created a multi-granular application layer (AaaS) and developed an application management platform (AaaS platform) between PaaS and IaaS. An application model and feedback model including performance metrics and classification rules were also introduced, as well as a parallel scheduling model and its policies. Based on a series of relevant definitions, fine-grained application units (Apps) can be combined to create multi-granular applications in the AaaS platform. These applications can be scheduled by a multi-core-aware parallel scheduling model, which not only increases system flexibility and applicability, but also improves the resource utilization of the fundamental infrastructure under fine-grained resource allocation.

Experimental results showed that based on application execution feedback on CPU cores and characteristics of multi-core processor architectures, the proposed multi-core-aware scheduling model can ap-

propriately assign applications to computing nodes in a balanced manner, fully utilize the CPU resources on computing nodes, and facilitate collaboration between computing nodes, CPUs on computing nodes, and CPU cores, yielding higher system performance and throughput than traditional methods.

Acknowledgements

This work is supported by the Funds of Core Technology and Emerging Industry Strategic Project of Guangdong Province (No. 2011A010801008, 2012A010701011, 2012A010701003), Guangdong Provincial Treasury Project (No.503-503054010110), Technology and Emerging Industry Strategic Project of Guangzhou (No.201200000034).

References

1. Armbrust M, Fox A, Griffith R, et al.: A view of cloud computing. Communications of the ACM, 2010, 53(4): 50-58.
2. Kang S, Kang S, Hur S.: A design of the conceptual architecture for a multitenant saas application platform. Computers, Networks, Systems and Industrial Engineering (CNSI), 2011 First ACIS/JNU International Conference on. IEEE, 2011: 462-467.
3. Azeez A, Perera S, Gamage D, et al.: Multi-tenant SOA middleware for cloud computing. Cloud computing (cloud), 2010 ieee 3rd international conference on. IEEE, 2010: 458-465.
4. Litoiu M, Woodside M, Wong J, et al.: A business driven cloud optimization architecture. Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, 2010: 380-385.
5. Lawton G.: Developing software online with platform-as-a-service technology. Computer, 2008, 41(6): 13-15.
6. Kibe S, Watanabe S, Kunishima K, et al.: PaaS on IaaS. Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on. IEEE, 2013: 362-367.
7. Kächele S, Domaschka J, Hauck F J.: COSCA: an easy-to-use component-based PaaS cloud system for common applications. Proceedings of the First International Workshop on Cloud Computing Platforms. ACM, 2011: 4.
8. Hu R, Li Y, Zhang Y.: Adaptive resource management in PaaS platform using Feedback Control LRU algorithm. Cloud and Service Computing (CSC), 2011 International Conference on. IEEE, 2011: 11-18.
9. Zhu J X, Bai W H, Li J G.: The mediation data services application model based on web resource ontology.//E-Business and E-Government (ICEE), 2010 International Conference on. IEEE, 2010: 1468-1472.
10. Zhu J X, Li G, Cai W W, et al.: Research of the Design and Application of the Negotiation Service Management Agent. Advanced Materials Research, 2011, 268: 1862-1867.
11. Dillon T, Wu C, Chang E.: Cloud computing: issues and challenges. Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on. Ieee, 2010: 27-33.
12. Vaquero L M.: EduCloud: PaaS versus IaaS cloud usage for an advanced computer science course. Education, IEEE Transactions on, 2011, 54(4): 590-598.
13. Zabolotnyi R, Leitner P, Hummer W, et al.: JCloudScale: Closing the Gap Between IaaS and PaaS. arXiv preprint arXiv:1411.2392, 2014.
14. Ferry N, Song H, Rossini A, et al.: Cloud MF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications. Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. IEEE Computer Society, 2014: 269-277.
15. Calin Sandru, Dana Petcu, Victor Ion Munteanu, et al.: Building an Open-source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources. 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing.
16. Lee J, Wu H, Ravichandran M, et al.: Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. ACM SIGARCH Computer Architecture News. ACM, 2010, 38(3): 270-279.
17. Bartolini S, Foglia P, Solinas M, et al.: Feedback-driven restructuring of multi-threaded applications for nuca cache performance in cmps. Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on. IEEE, 2010: 87-94.
18. Feitelson D G, Weil A M.: Utilization and predictability in scheduling the IBM SP2 with backfilling. Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998. IEEE, 1998: 542-546.
19. Liu X, Wang C, Qiu X, et al.: Backfilling under two-tier virtual machines. Cluster Computing (CLUSTER), 2012 IEEE International Conference on. IEEE, 2012: 514-522.