Characterization of Computational Grid Resources Using Low-level Benchmarks

George Tsouloupas Marios Dikaiakos
{georget,mdd}@ucy.ac.cy
Dept. of Computer Science,
University of Cyprus
1678, Nicosia, Cyprus

Abstract

An important factor that needs to be taken into account by end-users and systems (schedulers, resource brokers, policy brokers) when mapping applications to the Grid, is the performance capacity of hardware resources attached to the Grid and made available through its Virtual Organizations (VOs). In this paper, we examine the problem of characterizing the performance capacity of Grid resources using benchmarking. We examine the conditions under which such characterization experiments can be implemented in a Grid setting and present the challenges that arise in this context. We specify a small number of performance metrics and propose a suite of micro-benchmarks to estimate these metrics for clusters that belong to large Virtual Organizations. We describe GridBench, a tool developed to administer benchmarking experiments, publish their results, and produce graphical representations of their metrics. We describe benchmarking experiments conducted with, and published through GridBench, and show how they can help end-users assess the performance capacity of resources that belong to a target Virtual Organization. Finally, we examine the advantages of this approach over solutions implemented currently in existing Grid infrastructures. We conclude that it is essential to provide benchmarking services in the Grid infrastructure, in order to enable the attachment of performance-related metadata to resources belonging to Virtual Organizations and the retrieval of such metadata by end-users and other Grid systems.

1 Introduction

Information about the performance capacity of Grid resources[*] is essential for the intelligent allocation of resources to Grid applications. This need arises from the diversity in performance capacity, which is common-place in Grid environments. Performance capacity estimates can help users and schedulers make more informed resource allocation decisions by combining this information with information about application performance (empirical or other). It is also important for users to be able to justify their application performance running on a Grid resource, using low-level performance measurements of the resource.

In order to achieve our goal, which is to provide a better source of performance capacity information for Grid resources, a number of problems need to be addressed. First, we need to determine a *small* set of *basic* metrics reflecting the basic factors affecting performance of computational Grid resources. These metrics must undergo careful selection; they must be general enough to be applicable to heterogeneous resources, they must be simple, easy to understand and clearly defined, and they must be effective in the characterization of the basic factors affecting the *basic* performace of the resources. Once the metrics are established, the right benchmarks must be selected (or implemented) to deliver the metrics; the subtle differences of existing benchmarks and the different measurement methodologies need to be studied. The benchmarks must then be executed at the different Grid

[*]The term “Grid resource” and the term “site” are used throughout this article to refer to either 1) a set of machines (single, dual or quad CPU) in the form of a cluster, or 2) a shared memory multiprocessor (SMP).

resources of a Virtual Organization, both in a *periodic* and in an "*on-demand*" manner. *Periodic* execution is necessary as a means to make results readily available to schedulers or other services. *On-demand* execution is necessary for users who need to tune benchmark parameters. A user that needs to investigate specific attributes of a resource by specifying special parameters to a benchmark, e.g. specify a larger packet-size for a network benchmark, is given this opportunity through *on-demand* execution.

The benchmark execution is itself a complex task, as it has to address heterogeneous resources, using different Grid middleware (i.e. using different interfaces), with different configurations and policies. In the case of unattended periodic execution, the benchmark parameters (such as the number of CPU's) must be determined automatically, without user intervention. Once the measurements are gathered they need to be managed, stored and made available for retrieval by the decision maker, be it an end-user or a scheduler. Last, but probably the most difficult, is the task of interpreting the results and putting them to use in decision making.

In this article we address the issue of Grid resource characterization and it is important to mention that a necessary precondition for choosing a resource onto which to execute an application, is that the decision-maker has some understanding of the application's behavior (see [21, 19, 23]). For example, a tightly-coupled parallel application would require a resource (cluster or SMP) with a high-performance interconnect, as indicated by the appropriate metric. In the same manner, an application with high memory bandwidth requirements would better run on a resource that performed well on a memory bandwidth benchmark. Undoubtedly, most decisions will not be so trivial. Good decision-making will be based on the level of understanding of the application's behavior and the understanding of the provided metrics.

In the remainder of this article, Section 2 describes our approach to characterization of computational Grid resources through benchmarking, Section 3 provides a short description of the GridBench framework which we used to obtain our results and Section 4 describes the metrics and benchmarks which we propose for use in the characterization of resources. Section 5 presents the experiments we conducted and the last section we present our conclusions.

2 Resource characterization

In existing Grid infrastructures, the performance capacities of Grid resources can be obtained through Grid Information Services (such as the Monitoring and Discovery Service [6]). Scheduling decisions based on the performace capacity (or "speed") of the candidate resources have to rely, at best, on the size of main memory, number of CPU's, and their nominal speed (e.g. in MHz). This is due to the fact that this is what users or schedulers can expect from information services like MDS. Despite that, Grid information service designers recognizing the importance of performance capacity, have made allowances in their information schemata for including performance information. An example is the GLUE Schema [7], developed for interoperability between US and European projects (used by projects such as the EDG [14], CG [13], LCG [16] and iVGDL [10]), which includes placeholders for the SPECInt and SPECFloat benchmark metrics. An example of real MDS ouput is given in Figure 1. The extract shown in Figure 1 refers to the Grid resource "cgce.ifca.org.es".

Taking the example of the EU-DataGrid, EU-CrossGrid, LCG and other projects currently utilizing the "GLUE" schema for MDS, performance information (more specifically the SPECint2000 and SPECfloat2000 benchmarks) is provided as static data that needs to be determined and entered manually by the administrator during installation. This is shown in Figure 1 by the `GlueHostBenchmarkSF00` and `GlueHostBenchmarkSI00` attributes. Such information is potentially inaccurate because it is prone to human error, be it intentional or unintentional. The fact that this is static information also creates problems because experience shows that a resource's performance capacity does change over time, either by the addition or removal of CPU's or even by simple alterations in the configuration of the hardware or software.

Also, work has been done to "assess" the Grid using "probes" [3] but this work focuses mainly on file transfers, remote execution, and Information Service responses. Computational resource performance is not addressed.

```
dn: GlueSubClusterUniqueID=cgce.ifca.org.es,
        GlueClusterUniqueID=cgce.ifca.org.es,
        Mds-Vo-name=ifcapro,mds-vo-name=local,o=grid
objectClass: GlueClusterTop
objectClass: GlueSubCluster
objectClass: GlueSchemaVersion
objectClass: GlueInformationService
objectClass: GlueKey
GlueSchemaVersionMajor: 1
GlueSchemaVersionMinor: 1
GlueChunkKey: GlueClusterUniqueID=cgce.ifca.org.es
GlueHostApplicationSoftwareRunTimeEnvironment: CG2_0_4
GlueHostApplicationSoftwareRunTimeEnvironment: CROSSGRID
GlueHostApplicationSoftwareRunTimeEnvironment: LCG-2
GlueHostApplicationSoftwareRunTimeEnvironment: MPICH
GlueHostApplicationSoftwareRunTimeEnvironment: MPICH
GlueHostApplicationSoftwareRunTimeEnvironment: MPICH-G2
GlueHostArchitectureSMPSize: 2
GlueHostBenchmarkSF00: 328
GlueHostBenchmarkSI00: 409
GlueHostMainMemoryRAMSize: 627
GlueHostMainMemoryVirtualSize: 1144
GlueHostNetworkAdapterInboundIP: FALSE
GlueHostNetworkAdapterOutboundIP: TRUE
GlueHostOperatingSystemName: Redhat
GlueHostOperatingSystemRelease: 2.4.20-30.7.legacysmp
GlueHostOperatingSystemVersion: 1 SMP Fri Feb 20 10:12:55 2004
GlueHostProcessorClockSpeed: 1261
GlueHostProcessorModel: Intel(R) Pentium(R) III family 1266MHz
GlueHostProcessorVendor: GenuineIntel
GlueSubClusterName: cgce.ifca.org.es
GlueSubClusterUniqueID: cgce.ifca.org.es
```

```
dn: GlueCEUniqueID=cgce.ifca.org.es:2119/jobmanager-pbs-short,
                Mds-Vo-name=ifcapro,mds-vo-name=local,o=grid
    objectClass: GlueCETop
    objectClass: GlueCE
    objectClass: GlueSchemaVersion
    objectClass: GlueCEAccessControlBase
    objectClass: GlueCEInfo
    objectClass: GlueCEPolicy
    objectClass: GlueCEState
    objectClass: GlueInformationService
    objectClass: GlueKey
    GlueSchemaVersionMajor: 1
    GlueSchemaVersionMinor: 1
    GlueCEName: short
    GlueCEUniqueID: cgce.ifca.org.es:2119/jobmanager-pbs-short
    GlueCEInfoGatekeeperPort: 2119
    GlueCEInfoHostName: cgce.ifca.org.es
    GlueCEInfoLRMSType: pbs
    GlueCEInfoLRMSVersion: OpenPBS_2.4
    GlueCEInfoTotalCPUs: 20
    GlueCEStateEstimatedResponseTime: 0
    GlueCEStateFreeCPUs: 20
    GlueCEStateRunningJobs: 0
    GlueCEStateStatus: Production
    GlueCEStateTotalJobs: 0
    GlueCEStateWaitingJobs: 0
    GlueCEStateWorstResponseTime: 0
    GlueCEPolicyMaxCPUTime: 900
    GlueCEPolicyMaxRunningJobs: 2
    GlueCEPolicyMaxTotalJobs: 4
    GlueCEPolicyMaxWallClockTime: 7200
    GlueCEPolicyPriority: 1
    GlueCEAccessControlBaseRule: VO:cg
    GlueForeignKey: GlueClusterUniqueID=cgce.ifca.org.es
```

Figure 1: MDS output related to the resource “cgce.ifca.org.es”.

Characterization through benchmarking

Micro-benchmarks provide a commonly accepted basis for comparing different computer systems in terms of their performance. They are also used to investigate performance properties of computer systems under carefully tuned, benchmark-induced workloads that stress particular aspects of system performance. For example, CPU benchmarks such as *Whetstone* [4] focus strictly on CPU arithmetic operations in a tight loop with minimal memory requirements, thus disregarding the effect of other factors on overall performance. Micro-benchmarks have many uses to different kinds of users; For example, administrators could use benchmark measurements to detect and *pin-point* faults or problems in general, and end-users could use measurements to select appropriate resources for running their application.

A *small* set of suitable metrics is necessary for quantitative performance characterization. These metrics need to reflect the basic factors affecting performance of heterogeneous computational Grid resources. The size of this set, i.e. the number of metrics, should be of reasonable size. Too small a set would run the danger of missing essential performance factors while a large set of metrics would complicate decision-making by providing too much information, but more importantly it would impose a higher cost from the additional benchmark executions.

Once the set of metrics is established, a carefully selected set of benchmarks must be employed. It is also essential that the selected benchmarks run for the minimum amount of time and produce reliable results. While long-running codes (i.e, real or synthetic kernel-benchmarks) may potentially provide more accuracy or higher-level metrics, they are not desirable since they would tend to be more application-specific and they would incur a large cost for running benchmarks, especially if they are run at regular intervals.

The collected results must be made easily available for use by the decision-making process. The benchmarking process produces a high volume of measurement data, considering that some measurements are not simple scalars and that it would be of interest to maintain a historical record of performance measurements, e.g. for the statistical assessment of resource availability and dependability.

Filtering results using monitoring

Grid resources are shared and not under our control. For this reason it is important to know the state of the resource under measurement during benchmark executions. Monitoring of the resource under measurement can help assess the validity of a benchmark by providing insight regarding the conditions under which the benchmark was executed. The monitoring data should be collected and maintained together with the benchmark results. The benchmark results can then be "filtered" based on the results of monitoring, e.g. by dropping results where the monitoring indicates non-exclusive use of a machine.

3 GridBench

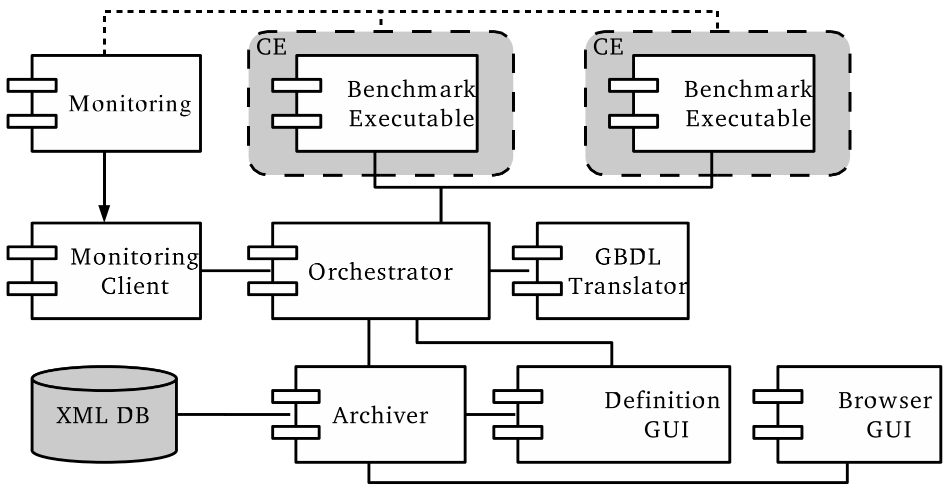


Figure 2: Architectural overview of GridBench.

GridBench [22] is a set of tools that aim to facilitate the characterization of Grid nodes or collections of Grid resources. In order to perform benchmarking measurements in an organized and flexible way, we provide the GridBench framework as a means for running benchmarks on Grid environments as well as collecting, archiving, and publishing the results. This framework allows for convenient integration of new and existing benchmarks into the suite, as well as the customization of existing benchmarks through parameters (e.g. specifying a different pattern for MPI communication).

Figure 2 is a diagram of the GridBench software architecture. The main components of this software architecture are:

- The *Orchestrator*: manages benchmark execution and collects results. It is accessible through a web-service interface;
- *Benchmark Components*: the benchmark executables, e.g. EPFlops;
- the *GBDL Translator*: converts XML descriptions of benchmarks to a Grid job description language. The current implementation can produce RSL (for Globus) and JDL (for EU-DataGrid/Condor);
- *Monitoring* Components: collect monitoring information from different monitoring services;
- the *Archiver* Database web-service: maintains benchmark results and makes them available through a web-service interface;
- *Benchmark Definition GUI* for defining and executing benchmarks; and
- *Benchmark Browser* GUI for browsing and analyzing benchmark results.

GridBench Components

The core functionality of GridBench is implemented by the following components:

The **GBDL Translator** has the task of parsing the benchmark specification written in the XML-based GridBench Definition Language (GBDL) (see [22] for more details). It then generates the job description language that is necessary for running the benchmark on a specific infrastructure (middleware). For prototyping purposes, the *Globus RSL* [5] and the *EU DataGrid JDL* [14], built on Condor-G classads [18], are included as output job definition languages.

The **Benchmark components** are the actual benchmark kernels. E.g. the STREAM [11] benchmark. They are adapted from existing benchmarks or implemented from scratch.

The **Orchestrator** component, which is implemented as a web-service, is responsible for coordinating the start-up and execution of the various components. On completion of the benchmark, the *Orchestrator* uses the *Archiver* web-service to store the resultant XML into a database.

The **Monitoring** component is a client to monitoring services. Monitoring of the resources under measurement is invaluable for understanding the results. The monitoring component takes a description of what to monitor through the GBDL and collects monitoring information. This is especially useful for "on-demand" execution of customized benchmarks (e.g. monitor memory usage and swap). The GBDL specifies the type of monitor (such as R-GMA [14] or JIMS [2]) and the target resource.

The results of the benchmarks are in the form of metrics. The metrics are incorporated back into the GBDL XML document along with the definition of the benchmark. The reason for keeping the specification and results together is that the benchmark results (especially for the more complex, multi-component benchmarks) make little sense without the specifications under which they were obtained.

4 Metrics and Micro-benchmarks

4.1 Metrics

A critical step in our methodology is the selection of a concise set of metrics for the low-level characterization of the Grid's computational resources. It is a reasonable assumption to make that the resource's performance depends mainly on the performance of its CPU's, the performance of its memory and caches, and the performance of its interconnects. Of course there is a wealth of other factors affecting machine performance ranging from I/O performance to Operating System robustness to fitness for running a specific application. We chose to limit the set of metrics to a concise size, but kept the design open for easy inclusion of more metrics as deemed necessary. In terms of specific metrics we have chosen (i) *Operations Per Second* for CPU performance (integer/floating-point), (ii) *Available Memory* and *Bytes per second* for writing and reading to and from main memory/cache, and (iii) *Latency* and *Bandwidth* for evaluating the machine's interconnects. These metrics are easily understood and well-established for evaluating their respective performance factor.

4.2 Micro-benchmarks

In order to deliver the required metrics, eight benchmarks are employed:
(i)*EPWhetstone*, (ii)*EPFlops*, (iii)*EPDhrystone*, (iv)*EPStream*, (v)*CacheBench*, (vi) *EPMemsize*, (vii)*MPPTest* and (viii) *b_eff_io*.

During the execution of each of the benchmarks listed above, it is imperative that the only process imposing substantial load on the CPU is the benchmark process, especially since the results are calculated using wall-clock time. Another thing to note is that the "EP" prefix of some benchmark names (namely EPWhetstone, EPFlops, EPDhrystone and EPStream) denotes the "embarrassingly parallel" nature of its execution, which means that each process runs on a CPU independently without any communication during the computation. The accumulated result from all the processes is then reported as the performance of the whole resource. In many cases it is useful to have results from benchmarks executed as both 1) one process per CPU and 2) one process per SMP node (see the description of *EPStream* for an example).

Table 1: Metrics and Benchmarks.

Factor	Metric	Delivered By
CPU	Operations per second (mixture of floating point and integer arithmetic)	EPWhetstone
CPU	Floating-Point operations per second	EPFlops
CPU	Integer operations per second	EPDhrystone
memory	sustainable memory bandwidth in MB/s (copy,add,multiply,triad)	EPStream
memory	Available physical memory in MB	EPMemsize
cache	memory bandwidth using different memory sizes in MB/s	CacheBench
Interconnect	latency, bandwidth and bisection bandwidth	MPPTest
I/O	Effective I/O bandwidth	b_eff_io

Table 2: Metrics returned by the "EPFlops" benchmark and the operation counts in each reported metric.

Metric name	FADD	FSUB	FMUL	FDIV	Total
mflops-1	21 (40.4%)	12 (23.1%)	14 (26.9%)	5 (9.6%)	52
mflops-2	58 (38.2%)	14 (9.2%)	66 (43.4%)	14 (9.2%)	152
mflops-3	62 (42.9%)	5 (3.4%)	74 (50.7%)	5 (3.4%)	146
mflops-4	39 (42.9%)	2 (2.2%)	50 (54.9%)	0 (0.0%)	91

EPWhestone is a simple adaptation of the traditional Whetstone CPU benchmark [4] so that it runs simultaneously on a set of CPU's using MPI. It is implemented in C, and uses MPI for collecting the final measurements from each process (communication time is excluded from measurements). Each process performs a mixture of operations, such as integer arithmetic, floating point arithmetic, function calls, trigonometric and other functions. The benchmark gets the current time using *gettimeofday()*, runs for a few seconds, calculates the wall-clock time difference and reports the rate at which these operations were performed on average. The typical execution time is less than 10 seconds.

EPFlops is a floating-point CPU benchmark adapted from the "flops" benchmark [1]. It is modified so that it runs simultaneously on a set of CPU's using MPI. It measures the performance of a CPU's floating-point operations in different "mixes" of floating-point operations. The benchmark employs a set of 8 modules, where each module is made up of a different mix of operations. Different combinations of the 8 modules yield a set of four metrics ("ratings") with different ratios of each of the four floating-point operations. The benchmark tries to maximize register usage in order to be as independent as possible from the performance of the memory sub-system. It is implemented in C. Table 4.2 gives a summary of the distribution of floating-point operations in the four metrics delivered by the "EPFlops" benchmark. For example, the *mflops-2* metric, which is also reported in Figure 6, does 152 operations per loop. Out of the 152 operations, 58 (38.2%) are additions, 14 (9.2%) are subtractions, 66 (43.4%) are multiplications, and 14 (9.2%) are divisions.

EPDhrystone is an integer operations benchmark, adapted from the C version of the "dhrystone" benchmark [24]. It is modified so that it runs simultaneously on a set of CPU's using MPI. Dhrystone is based on a workload from an extensive set of applications, but does not target numerical computations. It focuses on "systems programming" applications which perform mainly integer operations. As before, the benchmark has been adapted to run concurrently on a set of CPU's using MPI. The benchmark returns the accumulated result from all the processes in "dhrystones" per second.

EPMemsize is a platform independent benchmark that aims to measure memory capacity. It is written in C and it runs simultaneously on a set of CPU's using MPI. It first determines the maximum amount of memory that can be allocated. It then proceeds to determine the maximum amount of memory that can be allocated *in physical memory*. The size of physical memory available is important to memory-intensive applications that profit from allocating as much memory as possible while avoiding the use of slow swap memory. Detecting the physical memory in the machine in a platform-idependent way may not depend on any system-specific system call to get the memory size. More importantly, the value that is returned by a "get_free_memory()" system call is usually not the real amount of physical memory that can be allocated by an application; the system kernel, services as well as other processes also take up memory, filesystem caches etc. The benchmark operates by accessing memory until a substantial delay occurs (determined by a configurable delay threshold). The process is performed repeadedly and the maximum amount of memory allocated without incurring swapping is returned.

EPStream is a simple adaptation of the C implementation of the well-known STREAM memory benchmark [11] so that it runs simultaneously on a set of CPU's using MPI. The STREAM benchmark measures the sustainable local memory bandwidth (MB/s). It is a simple synthetic benchmark program and in addition to providing memory bandwidth it also gives an idea of the corresponding computation rate for simple vector kernels. The STREAM benchmark measures bandwidth while performing four operations: *copy*, *scale*, *sum* and *triad*. Table 3 outlines each operation. In the case of SMP machines, such as clusters of dual-CPU or quad-CPU machines, this benchmark can provide useful information when run in either of two modes: **1)** One process per SMP node (e.g. 1 process on a dual node) and **2)** One process per CPU (e.g. 4 processes on a quad node). This information can be crucial since the memory bandwidth available may be shared between more than one CPU's.[*]

Table 3: The STREAM benchmark Operations.

Name	Operation
copy	a[i]=b[i]
scale	a[i]=q*b[i]
sum	a[i]=b[i]+c[i]
triad	a[i]=b[i]+q*c[i]

The typical execution time of EPStream is around 10 seconds.

CacheBench is a benchmark aiming at evaluating the performance of the local memory hierarchy of a machine [12]. The benchmark is implemented in C and performs a set of operations – *read*, *write*, *read/modify/write*, *memset()* and *memcopy()* – varying the underlying array size thus exposing the performance of the (potentially multi-level) cache. For example, a knee can be observed at the different cache sizes when the results are plotted on a graph (see Figure 12). An instance of CacheBench is invoked on each CPU of the resource under study and results are reported independently for each CPU. The operations at each size run for a configurable amount of time (default is 2 seconds) and the average bandwidth (MB/s) is reported. Table 4 outlines each operation. While this benchmark produces a similar metric to the STREAM benchmark, it runs for a longer time since it takes measurements at different memory sizes (execution times are typically in the order of 5 minutes). The time it takes to finish depends strictly on the input parameters. It also focuses on the performance of memory *caches* providing insight to the different levels of cache available to the CPU's. Since this benchmark runs considerably longer than the EPStream benchmark, it would make sense to invoke it when need arises (i.e. the user is explicitly interested in cache performance, and the sustained memory bandwidth produced by EPStream is not adequate).

[*]An example of this is that Dual Intel "Xeon" nodes typically have a single memory controller per SMP machine, while AMD "Opteron" SMP machines typically have a separate controller for each CPU and can thus achieve a highter memory bandwidth.

Table 4: The CacheBench Operations.

Name	Operation
read	register=m[i]
write	m[i]=register++
read/write	m[i]=m[i]++
memset()	(system call)
memcopy()	(system call)

MPPTest is a benchmark that tests MPI communication speeds by various ways and provides a variety of options for a detailed performance analysis [9]. MPPtest is platform and MPI-implementation independent and can therefore be used with any MPI implementation. MPPtest aims to make reproducible measurements of MPI performance and results are claimed by the MPPTest creators to be reproducible since the reported measurements are the minimum of several runs. For the purpose of resource characterization it is desirable to have a focused set of measurements and to this end, only three types of measurement are performed: (i) Latency, (ii) point-to-point bandwidth and (iii) bisection bandwidth. "Bisection bandwidth" refers to an *all-to-all* measurement of bandwidth in contrast to the *point-to-point* measurement where only two processes communicate at any time. The typical execution time is in the order of minutes (depending on the measurement detail) and results are calculated using wall-clock time.

The b_eff_io benchmark is included in order to evaluate the shared I/O performance of (shared) storage at a resource (site). This benchmark is used "to achieve a characteristic average number for the I/O bandwidth achievable with parallel MPI-I/O applications" [17]. *B_eff_io* produces a metric given in Megabytes per second, which represents the average obtained by performing several storage access patterns. Access patterns include: (i)Multiple processes read/write data scattered in a file; (ii) Multiple processes read/write adjacent data; (iii) Multiple processes read/write data in separate files; and (iv) each of the multiple processes accesses data in a different segment of a segmented file (a detailed description of the access patterns can be found in [17]). Given that shared disk I/O is usually performed over the network, the results obtained by this benchmark may be correlated with the results obtained by the MPPTest benchmark. The benchmark is implemented in C.

5 Experimentation

The experiments described in this section were conducted on the EU CrossGrid testbed [8], which follows the architecture presented in Figure 3. In this architecture, a Grid Virtual Organization (VO) is made up of a set of geographically distributed sites (resources). Each site contains a Computing Element (the Gatekeeper in Globus terminology) which manages a set of "Worker Nodes". A site may contain a "Storage Element" which is an interface to mass storage. Typically, the Computing Element and Worker Nodes have direct (Local Area Network) access to mass storage on the Storage Element that is close to them (e.g. via Network File System). The Grid VO also contains some VO services such as a resource broker, VO membership server etc. The Sites are connected by shared wide-area links. (In our specific test-bed, sites are connected through the Géant [15] network).

The charts presented in this article are examples of GridBench-generated charts. They show benchmarks running at different resources on the EU CrossGrid testbed. These experiments were conducted using the *on-demand* paradigm described earlier, and it is the type of experiment that would help a user allocate the right resources for her application. The experiments were conducted using the GridBench framework where:

1. The user selects which benchmark to run;
2. The system determines the currently available resources by accesssing the Grid Information Service;

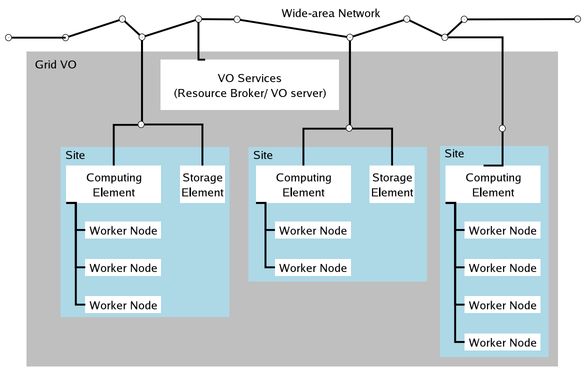


Figure 3: Basic Grid infrastructure architecture.

3. The system generates benchmark descriptions (see [22]) with appropriate parameters (e.g. currently available number of CPU's) for the target resources;
4. The system executes the benchmarks and archives the results; archived metrics are used to generate the charts;
5. The system collects monitoring information (by interfacing to external monitoring services) from the resource in question for the duration of the benchmark execution.

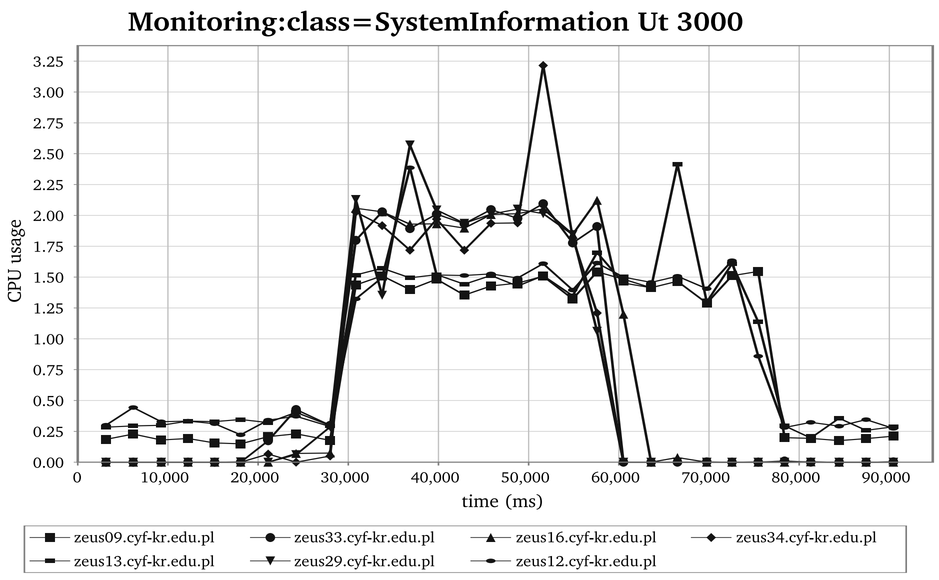
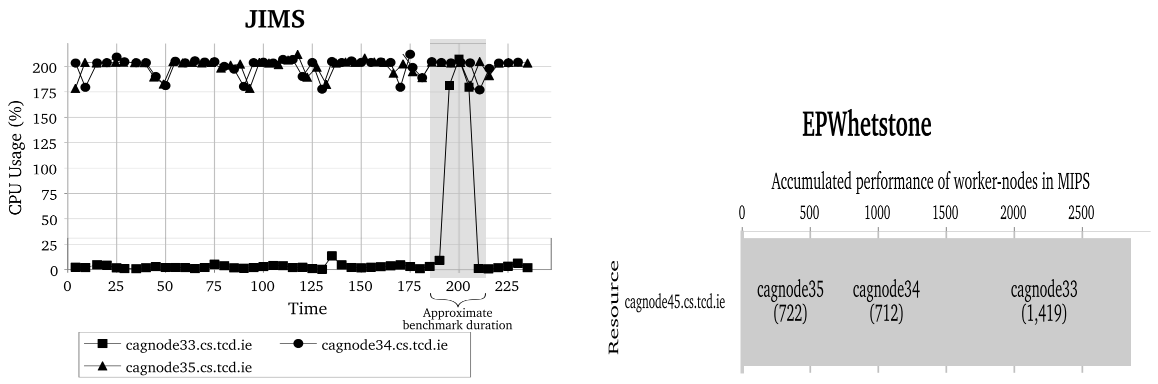


Figure 4: Monitoring of CPU usage on the resource during execution of the EPWhetstone Benchmark.

When a benchmark is performed using the GridBench framework, the user has the option of specifying what is to be monitored. This monitoring data is archived together with the benchmark results. During our experiments we monitored important machine parameters such as CPU load, memory usage and network load, in order to validate the benchmark results (e.g. by establishing that the benchmark had exclusive use of the machines on which it ran). For example, the monitoring information shown in figure 4 was collected during the execution of the EPWhetstone benchmark on the resource. Each curve on the chart indicates the CPU load on one worker node for a time interval that begins a few seconds before the benchmark execution and ends a few seconds after. (The CPU load is given as the ratio of time spent in user-mode to wall-clock time; values greater than 1 are due to the use of dual-processor machines). The obvious jump on the CPU load between seconds 30 and 80 was due to the load imposed by the benchmark. It is important for the evaluation of the benchmark result that the CPU load just before and right after the execution is negligible. While this is not proof that the benchmark was the only process putting load on the CPU, given the fact that the benchmark runs for a very shot time, it is usually safe to assume that if the CPU was idle just before and immediately after the execution then there was no other process imposing load on the CPU.



(a) CPU usage monitoring

(b) Benchmark result

Figure 5: A monitored execution of the EPWhetstone benchmark showing how the *invalid* results shown in (b) can be explained by CPU usage monitoring in chart (a).

Figure 5 shows an example of monitored benchmark execution where the results are invalid due to non-exclusive use of the resources by the benchmark. Figure 5(a) shows the CPU usage (automatically collected by GridBench) while Figure 5(b) shows the result. In Figure 5(a) there is significant CPU usage on two of the three worker nodes before and after the "grayed" area, which indicates the duration of the benchmark execution.[†] As a result, the measured performance of `cagnode34` and `cagnode35` severely suffers, as shown in Figure 5(b), due to CPU usage by other applications.[‡]

Figure 6 shows the result of performing the three CPU micro-benchmarks *EPDhrystone*, *EPWhetstone* and *EPFlops* on a set of resources. Quite apparently, the graphs are very similar. While the absolute numbers do change as far as the *EPDhrystone* benchmark is concerned, it is the *relative* performance that is important. The similarity is expected since all the sites participating in the testbed under study have pretty much the same type of processors (Intel PIII/P4), therefore the floating-point versus integer performance does not vary from site to site. It is also apparent that results from *EPWhetstone* and *EPFlops* are very close. Again this is expected since Whetstone relied more on floating-point operations than on integer operations. Since for the given set of resources there is no significant variance between results from the three benchmarks we select just one (*EPWhetstone*) for analysis.

Figure 7 shows results for the *EPWhetstone* benchmark, providing a "view" of the available resources at a point in time, from a CPU performance perspective. When EPWhetstone is executed on a Grid resource it returns a set of values, each value measuring the CPU performance of a single CPU.

[†]Values of 200% for CPU usage are due to the worker nodes having dual CPU's.

[‡]The benchmark was executed with two processes on each worker node

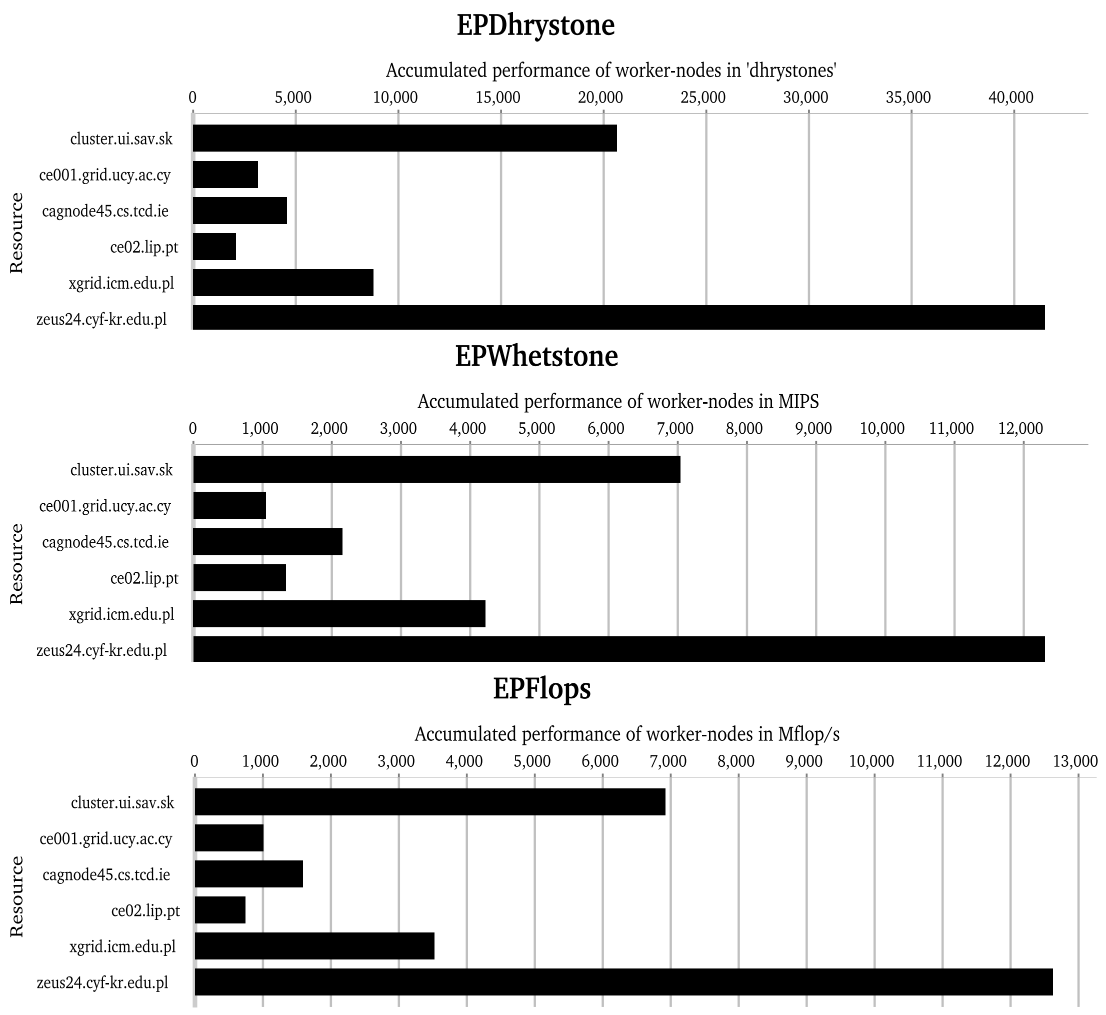


Figure 6: Accumulated performance of several clusters using different CPU performance metrics

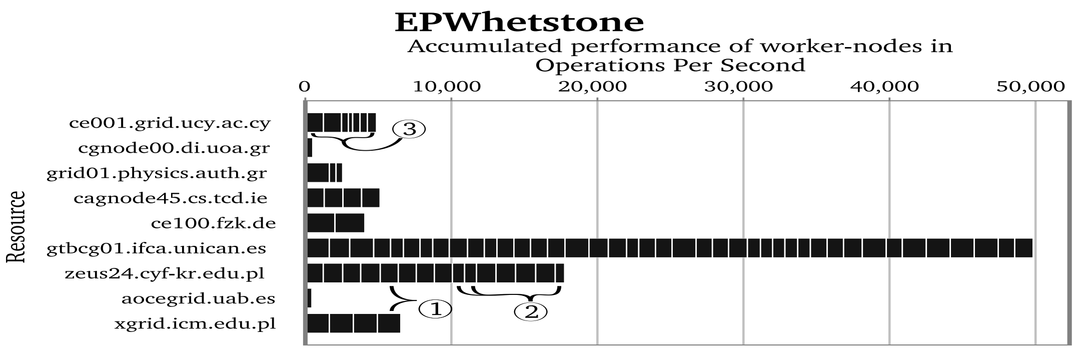


Figure 7: Whetstone performance of all computational resources currently available on the CrossGrid testbed (generated using the GridBench Browser)

When the resource under study is a cluster (as was always the case in our experiments) the hostname is returned along with the performance measurement. In the *stacked bar-chart* in Figure 7 each bar is made up of several segments. Each segment represents the contribution of a single cluster node. If a number of CPU's come from the same dual-CPU worker node, their performance is aggregated and displayed as a single segment.

The first, and probably most important, deduction to be made is that these resources *are operational*. This means that for each resource, the requested number of CPU's was allocated, all the processes were initiated and completed successfully and the results returned. Our experience indicates that job submissions do fail, and they fail *frequently*. There are many reasons for this, the most notable being the use of "stale" information obtained from Grid information services. For example, if the information service reports that a resource has more available (free) CPU's than it really does, then the job submission could fail or be postponed. Some data-intensive applications may require the staging of huge amounts of data, risking a big waste of time and bandwidth in the case when the application processes fail to spawn. Based on the success of the micro-benchmarks it is likely that if an application is submitted to one of these resources, its processes will be spawned successfully. A successfull benchmark execution, after reserving a resource and prior to staging and running the actual application, can verify that the resource is operational from a hardware and middleware point of view.

It is also evident in this chart that the different resources vary greatly in terms of processing power. They vary both in terms of the number of CPU's and in terms of individual CPU performance. For example, resource `zeus24.cyf-kr.edu.pl` has a larger number of CPU's than `xgrid.icm.edu.pl`, but the latter resource has faster CPU's. (See (1) in Figure 7). If we focus on resource `zeus24.cyf-kr.edu.pl` we can observe that 3 CPU's (indicated as (2) in Figure 7) appear to be performing slightly worse than the rest. This could be attributed to other processes running on the specific cluster nodes. This could also be attributed to other problems, ranging from hardware faults to software misconfigurations. It was in fact determined, by observing the monitoring information collected during the benchark execution, that on the three machines in question there was non-negligible CPU load right before and immediately after the execution of the benchmark. For some applications this could be of little importance, but for some tightly-coupled codes a single slow node could seriously impact performance. Internal resource uniformity can also be evaluated. Unevenly sized segments could result from a set of cluster nodes that are of dissimilar performance (e.g. (3) on Figure 7) where the resource is known to contain single-CPU as well as dual-CPU nodes.

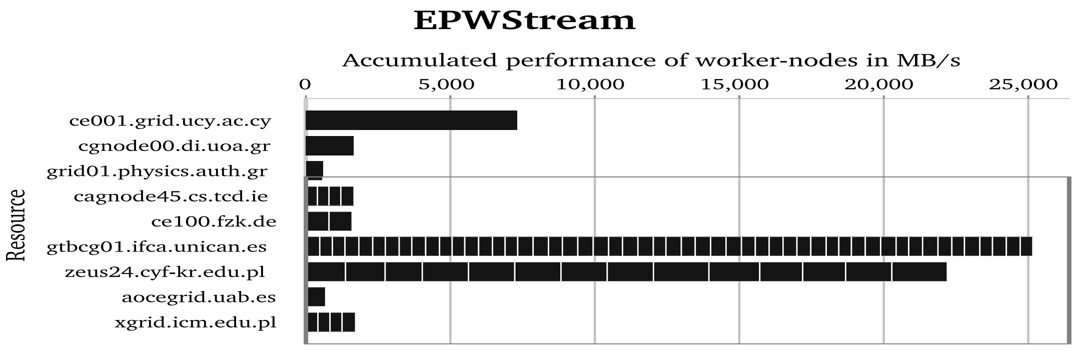


Figure 8: STREAM performance of all computational resources currently available on the CrossGrid testbed.

Figure 8 shows results for the *EPStream* benchmark, characterizing the Grid resources from a memory bandwidth perspective. Again, it is observed that memory performance of the resources is quite diverse. If the chart in Figure 8 is compared to the chart in Figure 7, it can be observed that the relative performances of the resources are in fact different when comparing based on memory performance rather than CPU performance. With memory intensive codes in mind, it would make sense for the user to make a decision based on memory bandwidth results rather than CPU results. In

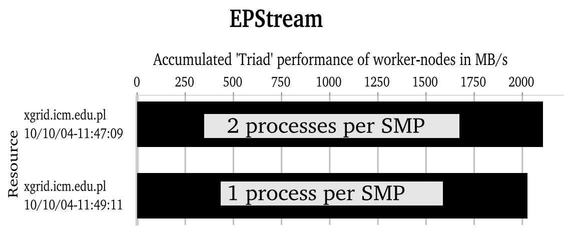


Figure 9: STREAM performance on a cluster of 4 dual-CPU worker nodes, using 2 processes per node (i.e. one process per cpu) *or* one process per dual-CPU worker node.

terms of internal resource uniformity, the same basic deductions described for the EPWhetstone results in Figure 7 apply. It is also notable that the large difference in CPU performance between resource `zeus24.cyf-kr.edu.pl` and resource `gtbcg01.ifca.unican.es` in Figure 7 is much smaller when comparing memory performance in Figure 8. A user intending to run a memory intensive code could choose to do so on resource `zeus24.cyf-kr.edu.pl` since it has good aggregate memory bandwidth using a smaller number of CPU's (potentially enjoying a speedup because of lower communication overhead). Figure 9 illustrates how memory bandwidth is shared between processes running on the same dual-CPU machine. The resource `xgrid.icm.edu.pl` provides 4 dual Intel PIII nodes. In this case memory bandwidth does not scale with the number of CPU's, in fact the aggregate memory bandwidth remains almost the same. This is another factor that could be taken into account when running memory-intensive codes.

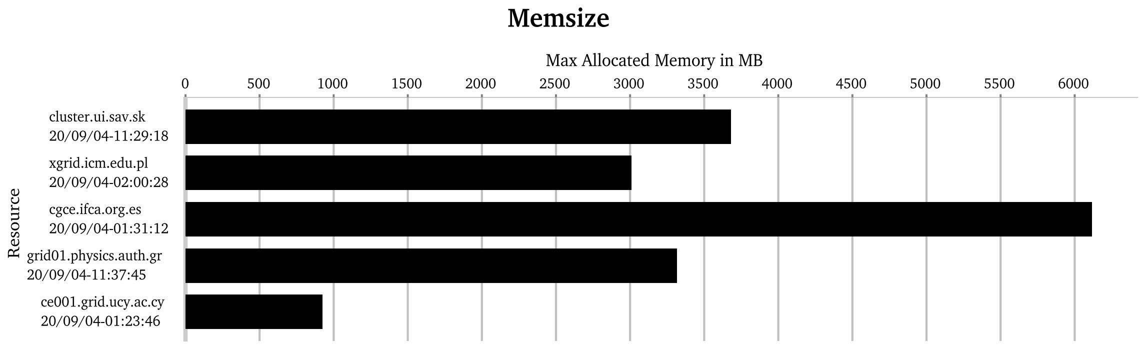


Figure 10: EPMemsize benchmark showing the approximate maximum amount of memory that could be allocated in physical memory.

In addition to memory bandwidth, the size of main memory is also important. Figure 10 shows the maximum amount of memory that could be allocated on the worker nodes in a set of resources. Lets take two examples: the resource *cluster.ui.sav.sk* having 16 nodes, each with 256MB without any swap (the site uses disk-less nodes) and *cgce.ifca.org.es* having a set of machines of 512 MB each and one with 1 GB. This information cannot be obtained from MDS. In the first case, an application can allocate less than approximately 220MB on each cluster node. In the second case an application can safely allocate close to 500MB with little chance of incuring swap memory. Therefore a memory-intensive application with a large memory footprint would consider sending a job to *cgce.ifca.org.es* over sending it to *cluster.ui.sav.sk*.

Figure 11 shows the point-to-point communication performace on four resources. Three of measurements coincide since the three sites employ the same network infrastructure, i.e. switched 100Mbit/s ethernet network. The fourth site *ce101.grid.ucy.ac.cy* has a 1Gbit/s network and performs significantly better (at least in terms of bandwidth. Similar (almost identical) results are obtained when

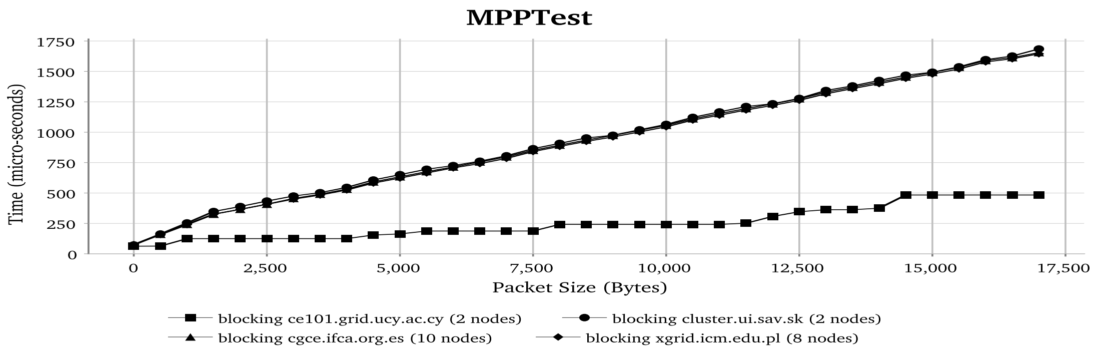


Figure 11: MPI point-to-point messaging performance on 4 resources. Three data series coincide since the three resources employ a similar networking infrastructure (100Mbit/s), while the other is significally different (1GBit/s).

measuring bisection bandwidth. The reason for this is that all the resources that were used in these experiments used switched networks, and none of them had a big enough number of nodes so that network performance degradation would manifest itself. The Latency is indicated by the time value of the zero-sized packet, i.e. the first data point on the graph, which is approximately 0.07ms for the sites with 100Mbit/s ethernet and 0.06ms for the site with 1Gbit/s ethernet. The bandwidth capability is indicated by the data points on the curve towards the larger MPI packet sizes. The bandwidth is calculated at approximately 8.7MBytes/s for the 100Mbit/s sites and at approximately 33MB/s for the 1Gbit/s site (33MBytes/s is possibly a result of a PCI bus bottleneck).

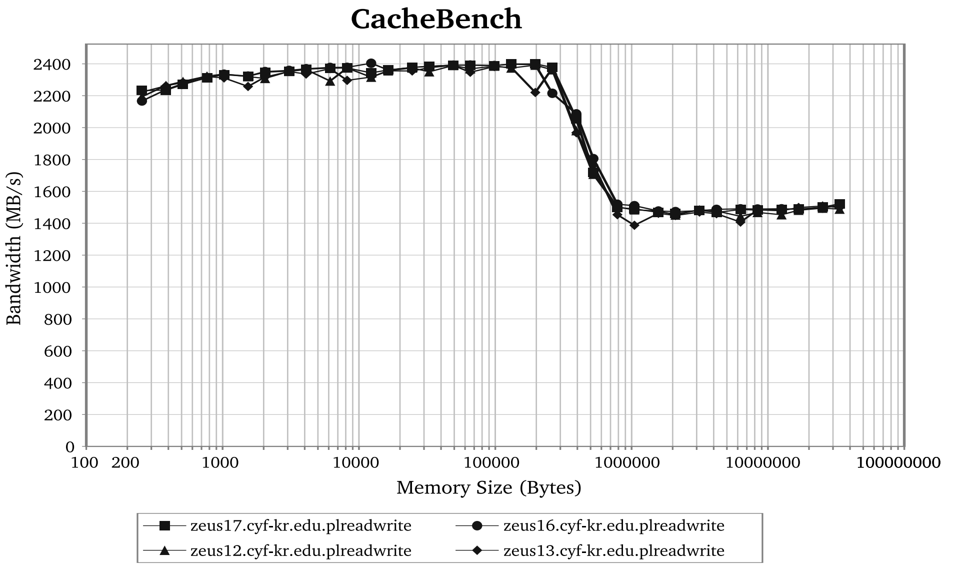


Figure 12: CacheBench benchmark showing the effect of the memory cache on memory bandwidth

Figure 12 shows the execution of the CacheBench benchmark on 4 CPU's on a resource. The specific metric is the "read/modify/write" metric giving the memory bandwidth in MB/s. The size of the memory used by CacheBench to measure the bandwidth is varied from 256 Bytes to 32 MBytes. The effect of the cache is apparent at the drop-off around 512 kBytes. A user could use this information to determine the cache size or verify that information provided by information services is accurate (when such information is provided). The user could also use this information to tune her application parameters for optimal use of the cache, of the specific site.

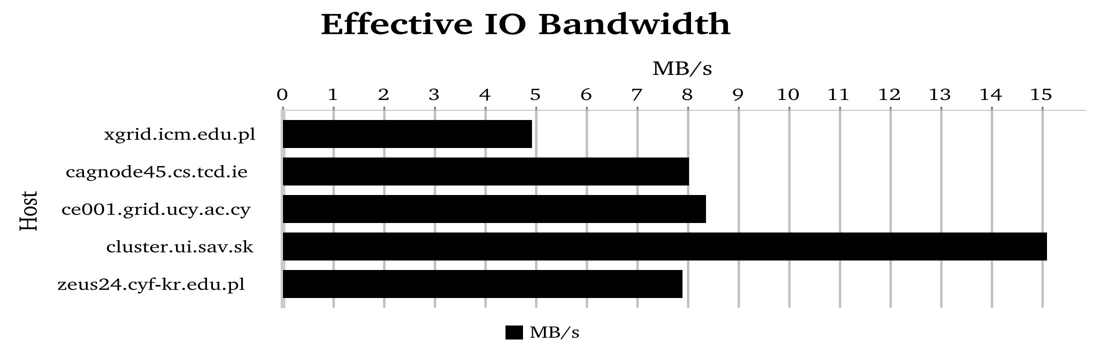


Figure 13: The *b_eff_io* benchmark on several sites on the testbed under. For each execution 2 CPU's (on separate worker nodes) were used

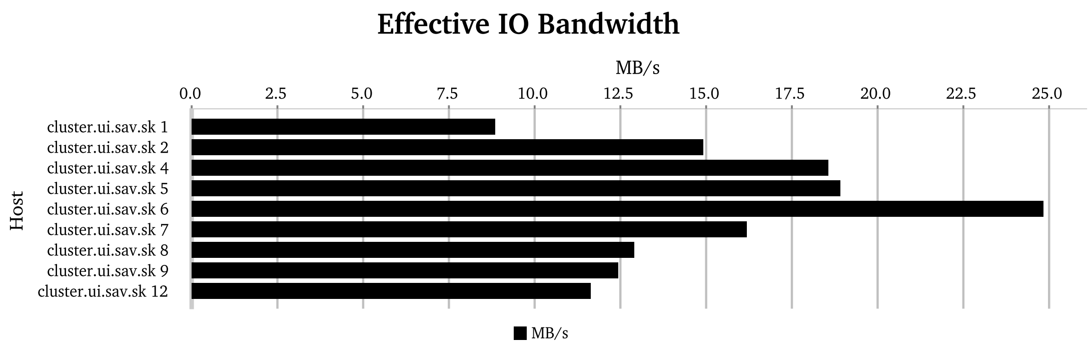


Figure 14: Varying the number of participating nodes for the *b_eff_io* benchmark on *cluster.ui.sav.sk.*

Figure 13 shows the results of the effective I/O bandwidth measurements on several resources. For each execution, 2 CPU's were used on 2 separate worker nodes. We chose to use CPU's on separate worker nodes since the access to shared disk would be over a Local Area Network. Having more than one process per worker node would (probably) mean that the processes on the same worker node (e.g. a dual-CPU worker node) would perform I/O over the same network interface card. We avoided this in order to keep result analysis less complicated. It can be seen that the resource *cluster.ui.sav.sk* performs considerably better than the others. We discovered that while the worker nodes were in fact connected over a 100Mbit network, the shared storage had a 1Gbit interface connected to the switch's 1Gbit uplink. To investigate this further, we performed a set of measurements (shown in Figure 14) on the specific resource. We varied the number of participating worker nodes and from those results it can easily be seen that the effective I/O bandwith of a resource varied considerably. Performance rises while going from 1 up to 6 worker nodes, probably due to parallelism, but drops off beyond 6 worker nodes, probably due to the storage device seeking too often trying to meet the requests from too many worker nodes. This could be taken further: by running enough tests, an "optimal" number of participating processes could be experimentally determined that is specific for each resource and depends on network connectivity, I/O device type etc.

All charts shown in this article were generated using the GridBench GUI using data archived in the XML database. This data is available for retrieval not only by end users, but also for automated decision-makers such as schedulers. A scheduler could use micro-benchmark results to *"rank"* the resources based on performance (CPU, memory or MPI). Additionally a scheduler could evaluate a resource's "health" by invoking one of the micro-benchmarks. Since execution times are typically less than 10 seconds, this would impose little additional delay and would potentially save a scheduler from time-consuming failed submissions.

Justifying application performance using micro-benchmarks

Figure 15 shows a set of charts in an attempt to show how a real application kernel's performance can be explained using micro-benchmark measurements on three Grid resources: 'cluster.ui.sav.sk" (*SK*), "cgce.ifca.org.es" (*ES*) and "xgrid.icm.edu.pl" (*PL*). *SK* has 12 CPU's, *ES* has 10 CPU's and *PL* has 8 CPU's. It is our purpose to compare the performance of the three sites "based on what they offer"; it is neither our purpose to compare the sites "on equal terms" (e.g. by using the same number of CPU's on each resource), nor evaluate the scalability of the code using different numbers of CPU's. This reflects scenarios where a user/broker has to make a choice of which resource to use. At a given point in time the choices could be: A) *SK* - 12 CPU's, B) *ES* - 10 CPU's and C) *PL* - 8 CPU's, while the decision could be affected by additional factors such as pricing. We will asses the performace of the three resources based on results from an application and then attempt to justify it with results from micro-benchmarks.

The application used, "bstream", is a blood-flow simulation code aimed as a pre-operative support decision system for vasular surgeons. The blood-flow simulation is part of an interactive Grid application that involves processing of 3D data obtained from MRI scanners, operation planning (such as a bypass), simulation, and finally blood-flow visualization [20]. Shown here is the computationally intensive part of the application, which is based on a lattice Boltzmann solver and uses MPI. This code was instrumented to measure elapsed time for each iteration, and integrated into GridBench.

Figures 15(c), 15(a) and 15(b), show iteration times (i.e. average time taken to complete an iteration of the blood-flow solver as the computation progresses), while the last six charts show results from all micro-benchmarks described previously. Looking at the iteration times chart (Figures 15(a) and 15(b)) we can compare performance of the application on the three chosen resources. *SK* is fastest of the three while *PL* is slightly faster than *ES*. *PL* with 8 CPU's is slightly faster (approximately 3%) than *ES* despite the fact that *ES* used 10 CPU's. The effect of varying the number of CPU's on this kernel is shown in Figure 15(c). Average iteration time does decrease as the number of CPU's increases (though not in a scalable manner) up to 12 CPU's (a limit imposed by the application). The sudden jump during the last iteration on each curve in Figures 15(c) and 15(a) is due to the "wrapping up" of the computation and the generation of output. It happens on the last iteration, regardles of how many oterations are chosen contributes very little when the ap

Once we analyze the performance of the application we proceed to analyze the micro-benchmark results. We can observe that the three sites *do* vary in their measurements, except for communication performance (Figure 15(g)).

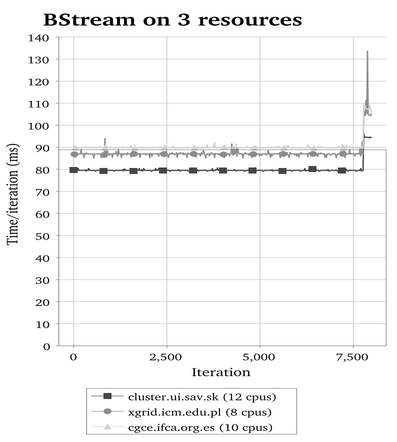
Network interconnect: In term of network interconnect performance the three resources show identical performance. This is as expected since the three sites employ a similar 100Mbit/s switched network.

CPU performance: Knowing that we will be looking for Floating Point performance in the application kernel, we can consider the EPFlops performance, shown in Figure 15(d). *Individual* worker node performance in *SK* and *ES* is similar while in *PL* it appears to be considerably better. This is in fact the effect of aggregating the performance of the two CPUs on dual-CPU worker nodes.

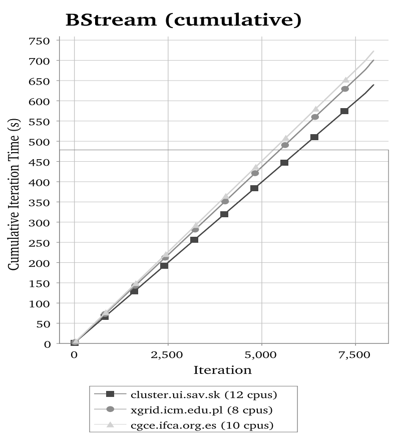
Memory and cache performance: As indicated by Figure 15(e), there is no significant variation on the *per-worker-node* memory bandwidth. In terms of memory size, Figure 15(f), the three resources again vary, with *SK* apparently having much less physical memory[§] than the other two. In terms of cache sizes, Figure 15(h), there is significan variation as indicated by the first "knee" on each curve. The curve for *SK* shows a knee at approximately 512KB while the other two show a knee at approximately 128KB[¶]. The values do not necessarily indicate the actual size of the cache,

[§] *SK* has 256MB of RAM (without any swap) on each of its nodes

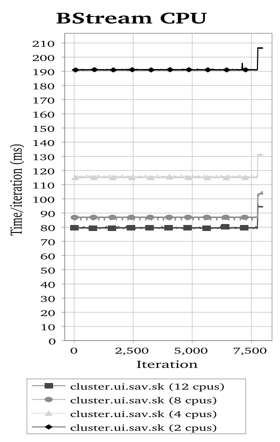
[¶] *SK* uses 1.8GHz Intel P4 CPU's while the other two sites use 1.266GHz Intel PIII CPU's.



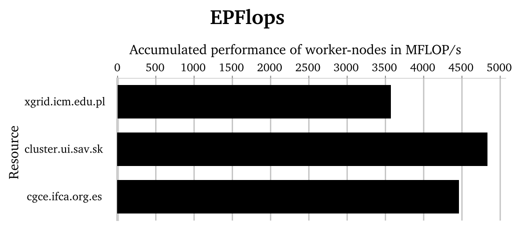
(a)



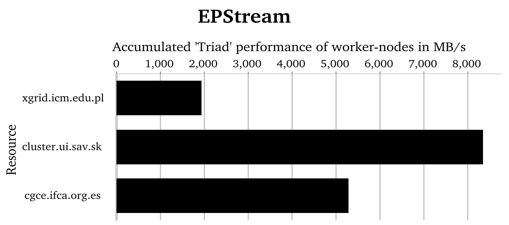
(b)



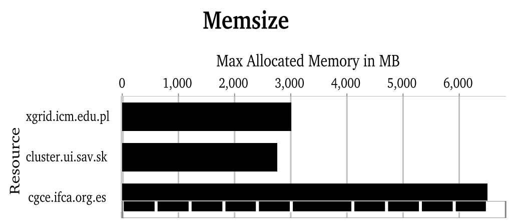
(c)



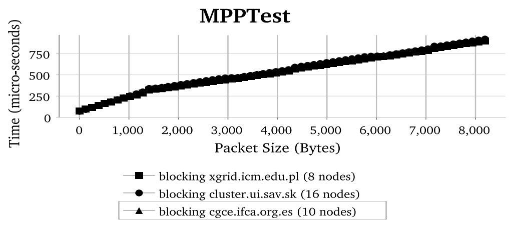
(d)



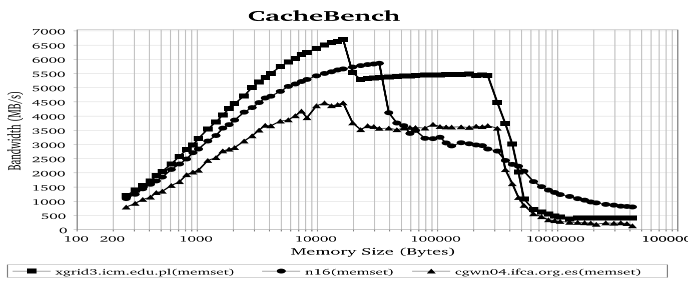
(e)



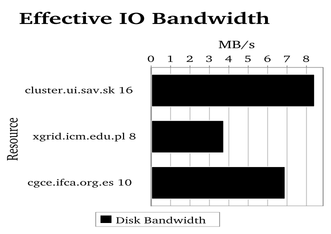
(f)



(g)



(h)



(i)

Figure 15: Bstream benchmark performance on 3 resources. Charts 15(c), 15(a) and 15(b) shows the average time per iteration as the computation progresses

but what is actually observed by the "triad" operation. Also noteworthy is that cache performance is approximately 50% faster at *PL*.

I/O performance: Figure 15(i) shows shared disk bandwidth where performance of the three resources *does* vary, but given that the application in question is not I/O intensive this will play little role.

The difference in performance between *PL* and *ES* (*PL* being faster) is probably *not* attributed to the difference in the number of CPU's, in fact the results should be the other way around. *ES* shows better aggregate performance than *PL* for CPU (EPFlops) and memory (EPStream), and it has more available physical memory (Memsize), yet it performs worse when running the "bstream" kernel benchmark. The underlying reason can probably be found in cache performance (shown in Figure 15(h)). *ES* cache performance is considerably less than that of *PL*, and *SK* displays a larger first-level cache than both *ES* and *PL*. At this point it is probably safe to assume that: 1) The better performance of "bstream" at *SK* is due to the additional number of CPU's; and 2) "bstream" performs better at *PL* than *ES* because of better cache performance.

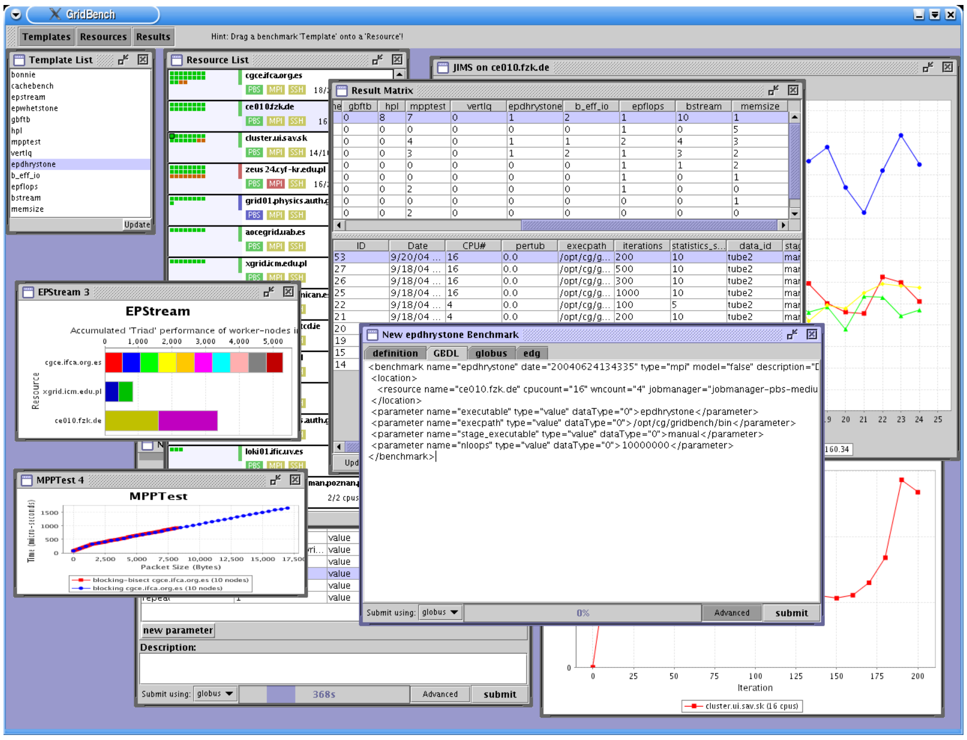
Once the relationship between the micro-benchmarks and the application kernel has been established it can then be applied to resource selection. From the findings above it could be concluded that *for this application* the decision maker (user or broker) should prefer cache performance over CPU count or CPU performance. A Resource Broker, scheduler, or end user can make more educated resource selections based on micro-benchmark results by taking the behavior of specific applications into account.

5.1 Practical Issues

During our experimentation and our effort to characterize a set of resources, we have come accross several issues regarding the functionality of the infrastructure. These problems fall in two main categories (i) configuration issues and (ii) general system issues. Configuration issues are administration-related and usually involve blunders or omissions by the site administrator, as well as conscious policy decisions that may cause problems. General system issues usually involve machines being in a erroneous state, run away processes that should have been removed, full disks etc. The fact that these problems surfaced, and could therefore be addressed, is another reason for running benchmarks. For example, early on in the experimentation there were issues regarding the execution of MPI codes. The errors were sometimes reproducible and sometimes not. By running microbenchmarks it was determined that some cluster nodes at several Grid sites were inaccessible (due to outdated OpenSSH keys), yet they were reported to be available in the local queues and the MDS. The administrators were contacted and the problem resolved. Another configuration issue that was detected at some sites by the execution of benchmarks was the incorrect spawining of processes. Specifically, more processes than available CPU's were spawned on a worker node, which was easily detected by looking at the results of CPU microbenchmarks. A problem that was often met and is categorized as a "gereral system issue" is the presence of run-away processes on several resources. This was detected by the observation of degraded performance by some microbenchmarks.

In many cases the underlying reason for failed benchmark executions or degraded performance of a benchmark was not determined, it is important though that many problems were detected and action could be taken to correct then. Some of these issues could have been detected by proper monitoring, but many of them would not surface without using an end-to-end test (involving most of the hierarchy of employed middleware) such as a benchmark.

Another issue we have come accross is database performance for storing the benchmark descriptions and results. One of the main reasons for using a native XML database was flexibility. Benchmark descriptions and results were in a semi-structured form and XML fits that purpose well. While performing the experiments it has been observed that the performance of the XML database degraded *significantly.* At one point the database was populated with approximately six thousand benchmark definitions and results. At that point, even simple queries took considerably long to execute; reply times were in the order of tens of seconds. Also, the fact that the data is in a semi-structured form makes it quite difficult to query the results. We plan to improve this in the future by specifying a more structured schema for benchmark results, which could even be implemented in a relational database providing speed and query flexibility.



he	gbftb	hpl	mpptest	vertlq	epdhrystone	b_eff_io	epflops	bstream	memsize
	0	8	7	0	1	2	1	10	1
	0	0	0	0	0	0	1	0	5
	0	0	4	0	1	1	2	4	3
	0	0	3	0	1	2	1	3	2
	0	0	0	0	0	0	1	1	2
	0	0	0	0	0	0	1	0	1
	0	0	2	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	1
	0	0	2	0	0	0	0	0	0

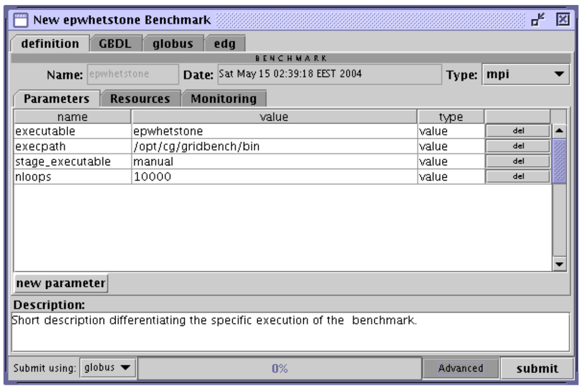
ID	Date	CPU#	pertub	execpath	iterations	statistics_s...	data_id	stag
53	9/20/04 ...	16	0.0	/opt/cg/g...	200	10	tube2	mar
27	9/18/04 ...	16	0.0	/opt/cg/g...	500	10	tube2	mar
26	9/18/04 ...	16	0.0	/opt/cg/g...	300	10	tube2	mar
25	9/18/04 ...	16	0.0	/opt/cg/g...	1000	10	tube2	mar
22	9/18/04 ...	4	0.0	/opt/cg/g...	100	5	tube2	mar
21	9/18/04 ...	4	0.0	/opt/cg/g...	200	10	tube2	mar
20								
19								
15								
14								

```
<benchmark name="epdhrystone" date="20040624134335" type="mpi" model="false" description="D
 <location>
   <resource name="ce010.fzk.de" cpucount="16" wncount="4" jobmanager="jobmanager-pbs-mediu
 </location>
 <parameter name="executable" type="value" dataType="0">epdhrystone</parameter>
 <parameter name="execpath" type="value" dataType="0">/opt/cg/gridbench/bin</parameter>
 <parameter name="stage_executable" type="value" dataType="0">manual</parameter>
 <parameter name="nloops" type="value" dataType="0">10000000</parameter>
</benchmark>
```

(a)

```
<benchmark name="epwhetstone"
           date="20040515023918"
           type="mpi" >
  <location>
    <resource name="cluster.ui.sav.sk"
          cpucount="16"
           wncount="16"
        jobmanager="jobmanager-pbs-workq"/>
  </location>
  <parameter name="executable" type="value"
         dataType="0">epwhetstone</parameter>
  <parameter name="execpath" type="value"
         dataType="0">/opt/cg/gridbench/bin</parameter>
  <parameter name="stage_executable" type="value"
         dataType="0">manual</parameter>
  <parameter name="nloops" type="value"
        dataType="1">10000</parameter>
 </benchmark>
```

(b)



New epwhetstone Benchmark

definition | GBDL | globus | edg

BENCHMARK

Name: epwhetstone Date: Sat May 15 02:39:18 EEST 2004 Type: mpi

Parameters | Resources | Monitoring

name	value	type	
executable	epwhetstone	value	del
execpath	/opt/cg/gridbench/bin	value	del
stage_executable	manual	value	del
nloops	10000	value	del

new parameter

Description:
Short description differentiating the specific execution of the benchmark.

Submit using: globus 0% Advanced submit

(c)

Figure 16: Defining benchmarks using GridBench: (a) A screenshot of the GridBench GUI showing the resource list, benchmark results and new benchmark defnitions, (b) a simple example of actual GBDL (GridBench Definition Language)
and (c) the definition panel.

6 Conclusions and Future Work

We have presented a concise set of benchmarks for the characterization of computational Grid resources, in terms of the performance of CPU, main memory and interconnects. We have also presented an adopted set of benchmarks to deliver those metrics. This small set of lightweight benchmarks can be run on the Grid resources with little overhead and with minimal effort by the user. The benchmarks can be invoked in a *periodic* and in an *on-demand* manner, using the GridBench framework. The resulting measurements are archived and made available via a web-service.

Low-level performance metrics, serving as metadata annotations to Grid resources, can be an aid for resource selection in the users' effort to "map" application kernels to appropriate resources. Micro-benchmark characterization of Grid computational resources can also be used by schedulers in the resource allocation process, as they can provide a basis for ranking resources using low-level performance metrics. Additionally, the execution of a micro-benchmark on a resource is in itself a validation of the operational state of the resource. Benchmarking services for Grids can play a really important role in tackling problems related to resource allocation.

We have also presented a set of results obtained from our Grid test-bed and described how results such as these can be useful. These results emphasize the variation of the performance capacity of Grid resources and the need to quantitatively assess the performance of each resource.

The use of benchmark results is not limited to just the ranking of resources according to performance (or detecting performance problems). An example would be the combination of these performance measurements with other external information such as resource pricing, which could prove quite useful. The low-level nature of the measurements makes no presumptions on the performance characteristics of any application, so these measurements could form the basis for a cost-model for charging for the use of computational resources. Furthermore, users could verify the "advertised" performance of a resource by running these light-weight benchmarks. Another example would be the administrative use of benchmarks to detect problems or faults in the Grid's computational resources.

In the future we plan to further investigate the relation between application performance and micro-benhchmark performance in the context of Grid environments. Building on the work presented in this article, we plan to further investigate the use of characterization in scheduling and resource allocation on the Grid. In addition we plan to investigate the use of micro-benchmarks for automated evaluation of Grid "resource health".

7 Acknowledgments

This work was supported by the European Union through the CrossGrid project (IST-2001-32243). The authors wish to acknowledge Alfredo Tirado-Ramos and Lilit Abrahamyan (University of Amsterdam) for the blood flow application code, and their support for running the distributed simulation.

References

[1] Al Aburto. flops.c version 2.0. ftp://ftp.nosc.mil/pub/aburto, 1992.

[2] Kazimierz Baos, Leszek Bizo, Micha Rozenau, and Krzysztof Zieliski. Interoperability architecture for grid networks monitoring systems. pages 245–253. Proceedings of Cracow Grid Workshop, October 2003.

[3] Greg Chun, Holly Dail, Henri Casanova, and Allan Snavely. Benchmark probes for grid assessment. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.

[4] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[5] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smithm, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–??, 1998.

[6] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–375. IEEE Computer Society, 1997.

[7] The Interoperability Collaboration (Grid Laboratory Uniform Environment GLUE). http://www.ppdg.net/mtgs/ivdgl/Interoperability.htm.

[8] J. Gomes, M. David, J. Martins, G. Tsouloupas, and M. Dikaiakos et. al. First prototype of the crossgrid testbed. 1st European Across Grid Conference, 2003.

[9] William Gropp and Ewing L. Lusk. Reproducible measurements of MPI performance characteristics. In *PVM/MPI*, pages 11–18, 1999.

[10] The iVDGL Project. http://www.ivdgl.org/.

[11] John D. McCalpin. *Sustainable Memory Bandwidth in Current High Performance Computers.* Advanced Systems Division Silicon Graphics, Inc., October 1995.

[12] Phillip J. Mucci and Kevin London. The cachebench report, 1998.

[13] The EU CrossGrid Project. http://www.eu-crossgrid.org.

[14] The EU DataGrid Project. http://www.eu-datagrid.org.

[15] The GÉANT Project. http://www.dante.net/geant/.

[16] The LCG Project. http://lcg.web.cern.ch/LCG/.

[17] Rolf Rabenseifner, Alice E. Koniges, Jean-Pierre Prost, and Richard Hedges. The parallel effective i/o bandwidth benchmark: b_eff_io. Message Passing Interface Developer's and User's Conference (MPIDC), March 2000.

[18] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.

[19] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, 1996.

[20] P.M.A. Sloot, A. Tirado-Ramos, A.G. Hoekstra, and M. Bubak. An interactive grid environment for non-invasive vascular reconstruction. In *2nd International Workshop on Biomedical Computations on the Grid (BioGrid'04), in conjunction with Fourth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)*, Chicago, Illinois, USA, April 2004. IEEE. CD-ROM IEEE Catalog # 04EX836C.

[21] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.

[22] George Tsouloupas and Marios D. Dikaiakos. Gridbench: A tool for benchmarking grids. In *Proceedings of the 4th International Workshop on Grid Computing (GRID2003)*, pages 60–67, Phoenix, AZ, November 2003. IEEE.

[23] Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance contracts: Predicting and monitoring grid application behavior. In *International Workshop on Grid Computing (GRID2001)*, pages 154–165, 2001.

[24] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.