Simulating Whole Supercomputer Applications

Juan Gonzalez, Marc Casas, Miquel Moreto, Judit Gimenez, Jesus Labarta, and Mateo Valero

Barcelona Supercomputing Center (BSC-CNS)/Computer Architecture Department - UPC
Barcelona, Spain
{juan.gonzalez, marc.casas, miquel.moreto, judit, jesus, mateo}@bsc.es

Abstract. Architecture simulation tools are extremely useful not only to predict the performance of future system designs, but also to analyze and improve the performance of software running on well know architectures. However, since power and complexity issues stopped the progress of single-thread performance, simulation speed no longer scales with technology: systems get larger and faster, but simulators do not get any faster. Detailed simulation of full-scale applications running on large clusters with hundreds or thousands of processors is not feasible.
In this paper we present a methodology that allows detailed simulation of large-scale MPI applications running on systems with thousands of processors with low resource cost. Our methodology allows detailed processor simulation, from the memory and cache hierarchy down to the functional units and the pipeline structure. This feature enables software performance analysis beyond what performance counters would allow. In addition, it enables performance prediction targeting non-existent architectures and systems, that is, systems for which no performance data can be used as a reference.
For example, detailed analysis of the weather forecasting application WRF reveals that it is highly optimized for cache locality, and is strongly compute bound, with faster functional units having the greatest impact on its performance. Also, analysis of next-generation CMP clusters show that performance may start to decline beyond 8 processors per chip due to shared resource contention, regardless of the benefits of through-memory communication.

1 Introduction

The current industrial trend towards large multi-core processors with multiple tens or even hundreds of cores poses an important challenge to research and development, both for hardware and software design. To date, researchers and developers do not have the right performance evaluation methods and tools to efficiently design hardware and software for large multi-core processor systems. The key enabler to performance evaluation in today's computer systems is simulation, especially in case real hardware is not (yet) available.

Architecture simulation is still a very sequential process, and so, it heavily depends on the processor's single thread performance. The advert of multicores has signaled the end of individual processor speedups, and has not produced any benefits to simulator performance.

The obvious solution would be to parallelize the simulators. After all, the hardware they are modeling is inherently parallel, and all of its components work seamlessly in parallel. However, the hardware parallelism is extremely fine grain, with synchronization happening among the different components at very high frequency (every nanosecond). Such synchronization has a significant hardware overhead that makes parallelized simulators spend more time in synchronization that on actual hardware modeling.

Since speeding up simulation by parallelizing the simulator is complex, the alternative is to reduce the amount of simulation performed, without losing relevance and accuracy. The most usual approach to reduce the amount of simulation is sampling. That is, the simulator proceeds through the application at fast speed without actually modeling the hardware, and at certain intervals (random, or carefully selected), changes to a highly detailed (and slow) simulation mode. Statistic analysis of this sampling approach shows that very high accuracy can be achieved through simulation of very small samples. Sampling has been widely used in the past for single processor simulation analysis.

However, when it comes to simulation of parallel systems and parallel applications, the sampling methodology presents several uncertain aspects: how should the application behave with regard to timing when it is not being modeled by the simulator? That is: how should the different application threads and hardware components align when the next detailed simulation sample is reached?

Our approach to this issue is to avoid modeling the most time consuming components in the multi-processor system: the processors and the cache hierarchy. In order to do that, we abstract the application in terms of a sequence of CPU bursts, and the communication and synchronization events that happen between them. Figure 2 shows an MPI message exchange phase segment of the NAS Parallel Benchmark BT with four threads.

However, even an abstracted application trace, and simulation restricted to MPI events represent a long simulation time. We have extended single-threaded simulation techniques to High Performance Computing (HPC) applications in order to further reduce the simulation time. Our full methodology is described in Figure 1.

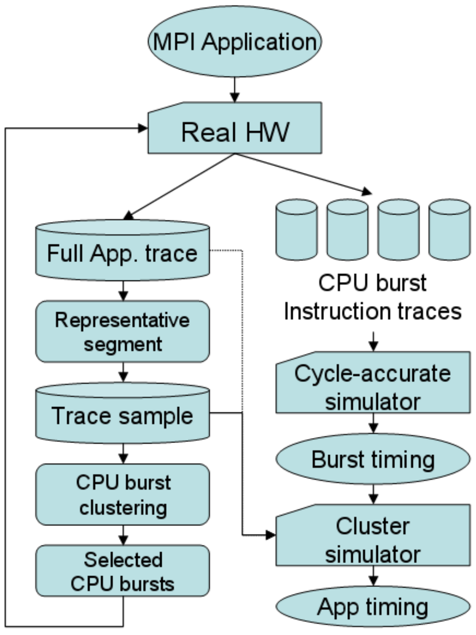


Fig. 1. Methodology cycle to perform whole application simulation

The methodology requires two executions of the application on real hardware. The first execution is instrumented at the higher abstraction level: CPU bursts, synchronization and communication events. It produces a full timestamped trace of events, annotated with hardware performance counters associated to each CPU burst. The full trace, representing hours of real execution, is still too large for processing. We apply non-linear filtering and spectral analysis techniques to determine the internal structure of the trace and detect periodicity of applications. Based on this analysis, we can cut a sample of the original trace, between 10 and 150 times smaller than the full trace [1], but still in the order of minutes of real execution and not affordable for a cycle-accurate simulator.

Next, we use a density-based clustering algorithm applied to the hardware counters values associated to each CPU burst (the region between communications). The clustering algorithm serves to identify the structural computation phases inside the period detected in the previous step. Each computation phase is composed by a set of CPU bursts identified as the same cluster. From the results of the clustering algorithm, we select a reduced set of representatives from each cluster.

Once individual CPU bursts have been selected, they are described to an instruction-level tracing tool that will run the application with minimal interference until the selected point in the selected threads have been reached. At that point, a detailed record of every instruction executed is stored to a file. Those instruction-level traces are then used for cycle-accurate simulation of the processor pipeline, and the cache and memory hierarchy. Finally, the timing information obtained from the cycle-accurate simulator can be fed-back into an application-level simulator that will rebuild the whole application timing accounting for the updated CPU burst duration, and maintaining the original synchronization and communication events.

The described simulation methodology is necessary to reduce the simulation time, either to obtain an estimation of performance on a different architecture (either at the cluster and interconnect level, or at the microarchitecture level), or to obtained detailed performance analysis data. As an example case of study, on a real application like WRF running with 128 threads, this methodology reduces the required simulation time by three orders of magnitude (5300x speed up). However we still need approximately half a day to estimate the speed of a given configuration performing all cycle-accurate simulations on a single processing machine. Accounting for the speedup factor, simulating the entire application at this level of detail would take over 100 days.

The rest of this paper is structured as follows. Section 2 discusses related work, and the contributions of our methodology over what was previously proposed. Section 3 describes in detail out full simulation methodology, from the contents of the application-level trace and the filtering and clustering algorithms, to the cycle-accurate simulator and how the timing information is re-introduced in the application-level simulator. Section 4 we provide examples on how we have used our methodology to produce both detailed performance analysis of full-scale HPC applications, and performance projections for next-generation HPC cluster architectures based on single-chip multiprocessors. Finally, in Section 5 we summarize our results, and present our conclusions regarding the discussed simulation methodology.

2 Related Work

Simulation tools have been widely used to verify, analyze and improve computer systems. Simulation is used at different levels of detail, depending on the particular target system to study. The tradeoff between simulation speed and accuracy is always present in these studies. As far as we know, no other current simulation infrastructure allows the simulation of large-scale computing systems like supercomputers at the same level of detail provided by our methodology.

Functional simulators emulate the behavior of the target system, including the OS and the different devices of the system (memory, network interfaces, disks, etc.). These simulators allow designers to verify the correctness of systems and develop software before the system has been built, but the real performance of the system cannot be estimated with them. Some examples are SimOS [2], QEMU [3] or SimNow [4].

Microarchitecture simulators model in detail the architecture of the processor and can estimate the performance of an application with different processor configurations. Simplescalar [5], SMTSim [6] and Turandot [7] are examples of this kind of simulators. However, these simulators normally do not model the interaction between the architecture and the OS and other system devices.

Full system simulators include the features of functional and microarchitecture simulators at the cost of simulation time. Some examples are SimICS [8] and COTSon [9]. Simulating a single processor with these simulators is very time-consuming, which makes unaffordable to evaluate supercomputers with hundreds of processors. To solve this problem, sampling techniques and extremely simple processor models have to be used [9].

In Carrignton et. al. [10] approach, they make the assumption that the performance of HPC applications can be almost totally explained taking into account the single processor performance of the application and its use of the network. From this starting point, they develop a framework which takes into account two kinds of data: First, they use a machine profile, that is, a characterization of memory performance capabilities of the machine and a characterization of its network performance. Second, they use an application signature, that is, a characterization of memory and networks operations needed to be performed by the application. Information related to memory of the machine and the application is correlated using Metasim [11]. Information related to network of the machine is correlated using Dimemas simulator [12]. Finally, they obtain a performance prediction of the application on the machine. The accuracy of this scheme is reported to be quite good.

However, this approach has several issues that have to be addressed: First it allows to make predictions only on existing systems because a machine profile is needed in order to obtain the predictions. Second, the Metasim simulations can be very expensive. Third, the flexibility of the ratios between the target machine and the initial machine is small because rates can be applied over the whole execution or over user functions.

Our approach allows performance simulation for not available architectures, and reduces remarkably the simulation time. Also, using our approach, it is possible to apply these ratios in a more fine-grain level. In conclusion, our approach takes into account the experience of [10] one but it improves remarkably important issues such us the flexibility, the compute time required to obtain the predictions, and the accuracy of the results.

3 Simulation methodology

In this section we describe in detail the different steps required to perform a detailed simulation of a large-scale MPI cluster.

To illustrate the proposed methodology, we will use an example trace of NAS BT class C parallel benchmark with 4 MPI tasks. See section 4.1 for further information about this benchmark.

3.1 Application level trace

The first step is to obtain a trace of the entire application at the level of abstraction of the programming model primitives. To obtain this trace, we use the MPITrace [13] tracing library. This library, using an interposition mechanism, is able to capture all MPI calls. In addition, the library can read the values of hardware counters using the PAPI library. As a result of executing an MPI application using MPITrace we obtain a time-stamped trace of the parallel application. The most important records of this trace are:

- States representing regions of time. Used to mark where a task is performing a concrete task (e.g. computing, waiting a message, etc.)
- Events containing a pair Type/Value. Used to report specific situations on the execution, e.g. hardware counters values.
- Point-to-point communications between different tasks. Used to report the amount of data transmitted between tasks.

To visualize this trace we use Paraver[13]. This tool is able to visualize the huge amount of information of the trace in a clear time-line. Figure 2 shows an example of Paraver time-line. In this figure, we can see a region of the execution of the NAS Parallel Benchmark BT with four tasks. A blue section represents a computation region. The rest of colors represent different communication primitives. Finally, the yellow lines represent point-to-point messages sent between different tasks. In following sections, we use different Paraver time-lines to illustrate the methodology of our study.

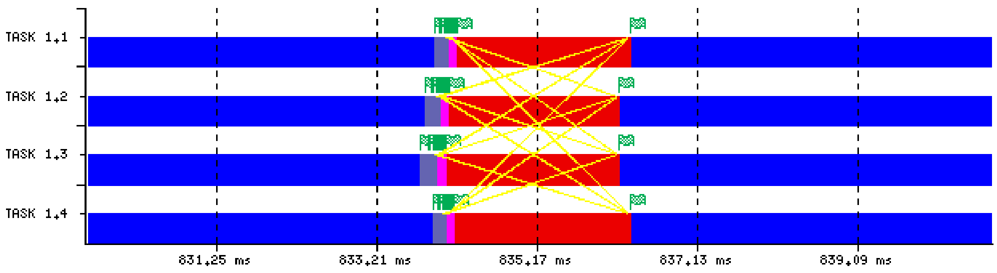


Fig. 2. Example of Paraver time-line of a region of NPB BT Class A with 4 tasks

3.2 Representative segment

Simulating the entire application is not feasible. However we can benefit from the repetitive nature of HPC applications, and select a representative segment of the whole application run, containing one or more iterations of the main loop. It is important to state that we work with complete iterations of the main loop. This is because the span of the representative segment is not a parameter of the analysis. Other tools, such as SimPoint [14] need the length of the representative segment as an input parameter.

Detection of the Periodic Phases We use signal processing techniques to automatically detect periodic phases. The Discrete Wavelet Transform (DWT) is applied to the signal obtained from the trace file. DWT has several desirable properties: First, it can be computed in $O(n)$ operations. Second, it captures not only the values of the frequencies of the input signal but also the physical location where these frequencies occur. These two properties are very useful because they allow us to find different execution phases given a trace file with only $O(n)$ operations.

However, the main problem of DWT is the accuracy. For low frequencies, it captures very well their values but fails to accurately capture the physical location where these frequency occur. Besides, for high frequencies it captures very well the physical location but fails on the frequency values. For a further approach to DWT and how we use it, see [15].

If the execution that the automatic system is analyzing has the typical structure of HPC applications, the periodic phase will be detected because it has a strong high frequency behavior. On the other hand, if the execution contains multiple periodic phases, the automatic system will also detect them, if these multiple periodic phases have a high frequency behavior.

On the top of Figure3 we highlight the periodic region identified by the wavelet transform. We can also see how initial and final phases, ruled out by wavelet, do not have periodic behavior. The analysis is based on signals extracted from the trace files. How are extracted these signals and what kind of information is contained on them is explained in [1] and [15].

Study of the periodic phase The periodic phase typically shows a pattern repeated many times. Let's assume that this pattern has a time span of $\frac{T}{2}$. Since there are no significant differences between the repetitions of the pattern, it is necessary to select several of them to simplify the subsequent performance analysis. On the other hand, it is possible that each of these iterations have several periodic regions within it. These periodic regions correspond to internal subloops of the main loop. Therefore, there is a need for a hierarchical search for periodicity over the periodic region.

At the end of the automatic analysis, for each periodic region detected, a subtrace which contains two iterations is generated from the original trace file. This subtrace starts at time t and finish at time $t + T$

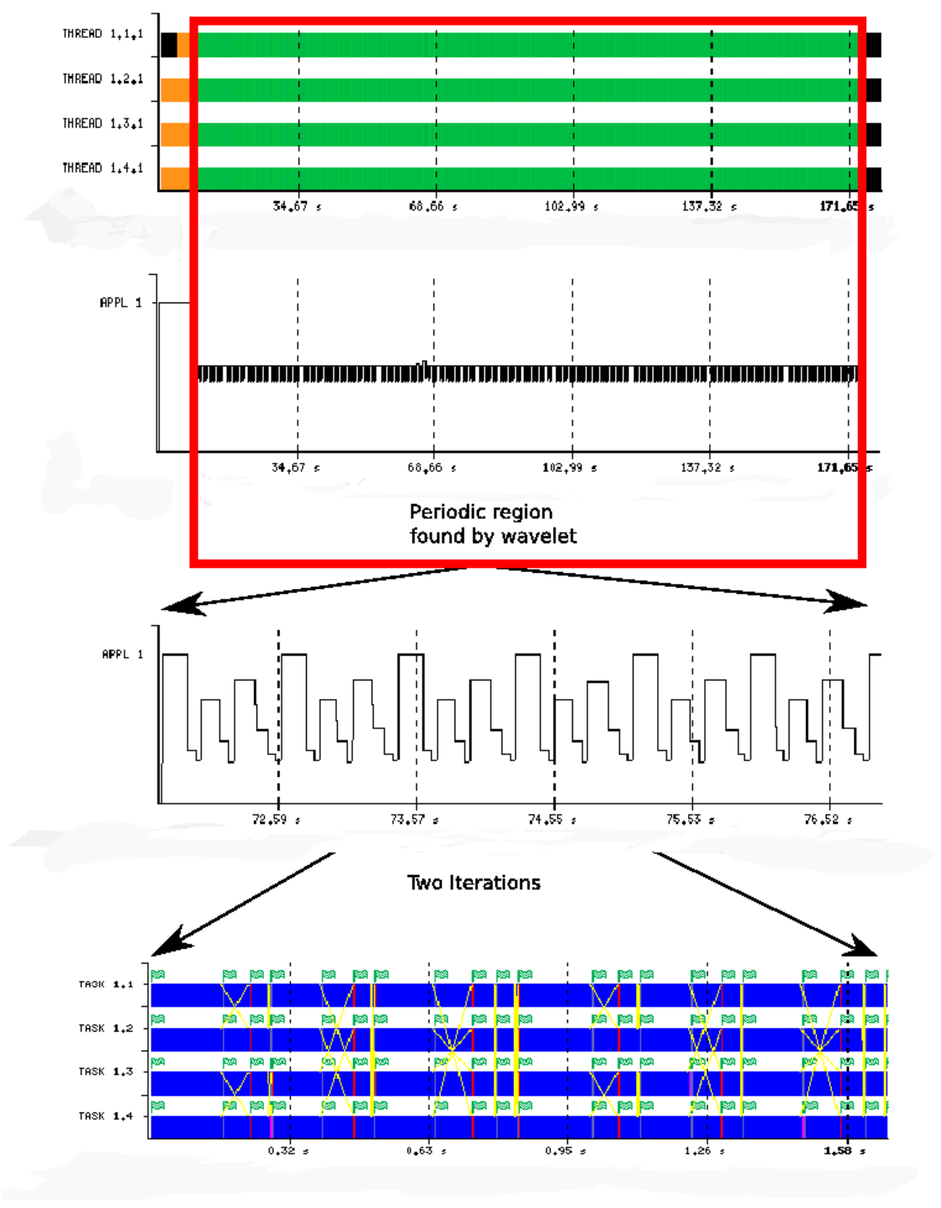


Fig. 3. Application of the size reduction. The input is a trace files obtained from the execution of NAS-BT class A application on 4 processors in MareNostrum supercomputer. First, the automatic system derives a signal from the trace file. After that, Wavelet Analysis finds the periodic phase and, finally, a subtrace that contains two iterations is generated.

. In Figure3, we show two periods detected by the algorithm. The extraction of hierarchical patterns and the detection of their time span is detailed in [1].

3.3 Computation bursts characterization

Once a representative segment of the application has been selected, we must characterize the different CPU bursts present in such a segment.

In order to perform this characterization, we apply a cluster analysis of this segment. This cluster analysis is described in [16]. It consists in detecting the similarities of different CPU bursts in terms of hardware counters metrics.

The results of applying the computation bursts characterization show us the structural similarities of computation bursts among all application tasks, i.e. the computation structure of the application. In fact, the underlying idea of this method is similar to the phase detection, but, at a finer granularity level.

Data Preparation The data used to characterize the application is the hardware counters provided by modern processors. The performance counters are read when each CPU burst finishes, in other words, just before a communication is executed. In order to optimize the results of the algorithm, we perform a simple preparation of the data.

Data pre-processing is the first step to reduce the volume of the clustering algorithm input data. We apply two different techniques: first, a filtering stage to discards those CPU bursts which add no information to algorithm; second, a logarithmic and range normalization of data, so as not no bias the clustering results when using hardware counters with different ranges.

Dimensionality reduction is a common problem when applying clustering algorithms. For example, in a PowerPC 970 platform, up to 8 different performance counters could be associated to each CPU burst, leading to an intractable 8-dimensional graph, in terms of algorithm complexity. Our proposal to address this problem is to reduce the dimensionality by selecting counters or derived metrics with "physical" meaning to the analyst, as those proposed in [17]. In our experiments, two different groups of the available metrics were used:

– Processor Cycles combined with IPC. This combination focuses the clustering on the *performance view* of the application.
– Completed Instructions , L1 and L2 cache misses. This combination reflects the impact of the architecture on the application structure, via the cache misses counters.

These two combinations are useful to detect regions with different computational complexity (Instructions Completed), and at the same time to differentiate between regions with the same complexity but different performance.

Clustering algorithm The clustering algorithm used in this study is DBSCAN [18]. (Density Based Spatial Clustering of Applications with Noise). This is a representative algorithm of the density-based clustering family. We selected this kind of algorithms due to the no data model assumption. K-means-like algorithms assume an Gaussian spherical shape of clusters but the hardware counters data distribution is not always distributed in this way.

The inputs of DBSCAN are two parameters, the radius Epsilon (*Eps*) and minimum number of points (*MinPoints*) plus the data itself. The resulting clusters obtained are those subsets C_i of the data that fulfill the following [1]:

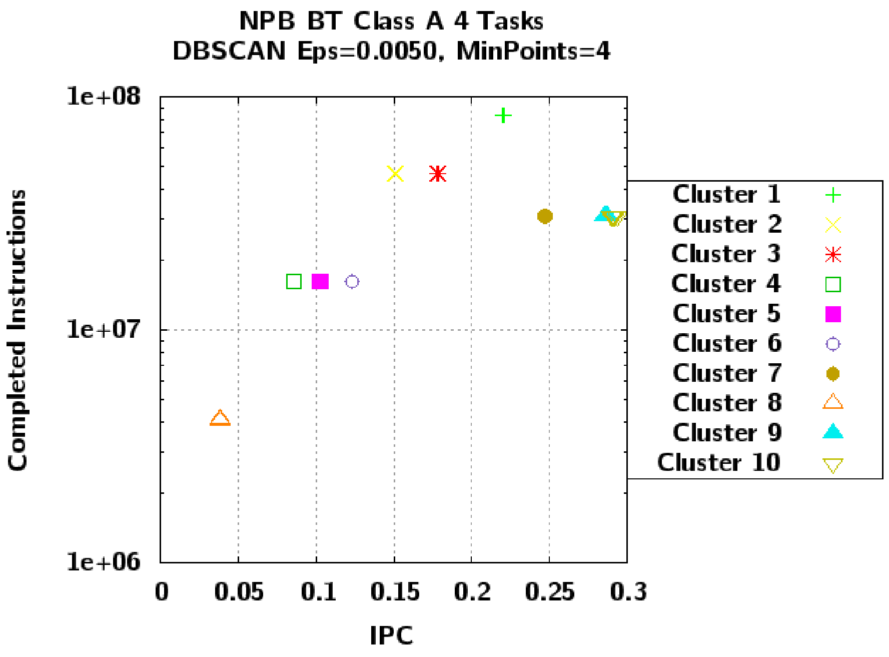
– For any given pair of points $p \in C_i$ and $q \in C_i$ it is possible to find a set of points $a_1, a_2, ..., a_{n-1}, a_n \in C_i$, being $p = a_1$ and $q = a_n$, where the Euclidean distance for each pair (a_i, a_{i+1}) is less or equal to *Eps*. This property is called density reachability.
– $|C_i| \geq MinPoints$. This is the minimum density condition to consider C_i as a cluster.

The technique applied for parameter selection is also described in [18] and it consists of generating a histogram with the sorted k-neighbor distance, being k the desired value of MinPoints. Then this distance is sorted (descending) and plotted. The histogram will show a descending curve. In [18], the authors suggest that the optimum value of *Eps* is the distance where the curve makes its first inflexion

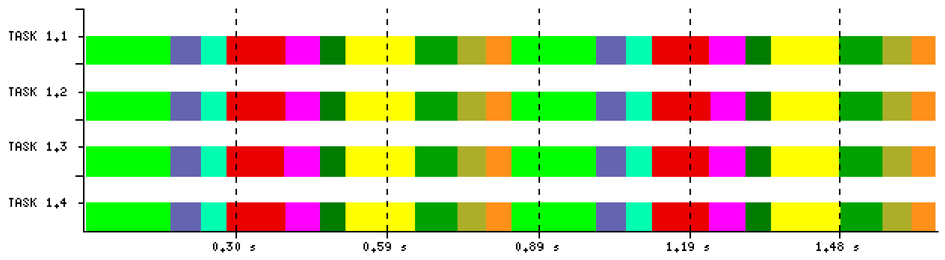
[1] These definitions are not exactly the same as those found in [18], but easily show the algorithm basis

(or "valley"). The points located on the left of this "valley" will be noise in the resulting partition and the rest will be present on one cluster. In [18], the authors ensure that choosing 4 as the default value of *MinPoints* produces the best results in 2-dimensional clustering. In our experiments higher values, usually 10, obtained a better characterization of applications structure.

Clustering results Once the clustering algorithm is applied, we obtain a Paraver trace where each bursts is marked with the cluster identification it belongs to. In addition, the clustering tool also generates a scatter plot of clustering dimensions (i.e. the hardware counters used) is depicted and the points are colored with the cluster assigned. Figure 4 presents the results of applying the clustering to the segment obtained in the previous step.



(a) **Scatter-plot of Completed Instructions vs. IPC, showing the resulting clusters**



(b) **Time-line of cluster distribution during the application execution**

Fig. 4. Results of applying DBSCAN clustering algorithm to the representative segment.

3.4 Cluster Representatives

After the CPU bursts characterization, we select a reduced number of representatives from all the clusters that represent a significant percentage of the execution time of the application. Only these cluster representatives will be simulated at the microarchitecture level. We consider the minimum number of clusters that cover more than 80% of the total execution time of the application. In the applications analyzed in this work, this step filters the number of clusters to less than six. We assume that the remaining clusters will show the same average behavior than the selected ones.

The selection of the cluster representatives is the step that requires more human intervention. We have tried different approaches to select a small subset of tasks (from 1 to 5) for each cluster: random, based on the duration, centroid of the cluster, etc. Once the subset of tasks has been selected, we need to select one (or multiple) instances of the corresponding cluster. Notice that each cluster is executed several times during the execution of each task. To simplify, for each pair (task, cluster) we select the first instance within the subtrace of the representative segment of the parallel application. To identify the first cluster representative executed by this task, we need to compute the number of the reference of MPI call from the beginning of the execution. This step requires filtering and extracting data from the original huge trace file. The references of the remaining cluster representatives are obtained relative to the previous cluster and only require working with the representative segment. Up to now, the experiments indicate that the selection criteria does not significantly affect to the accuracy of this methodology.

3.5 Microarchitecture Level Instruction Trace

Once the representatives have been selected, we have to obtain a microarchitecture level trace, at the instruction level. We have used `valgrind` [19], a dynamic binary instrumentation tool, to trace each cluster representative of the parallel application. The same information can be extracted with other dynamic instrumentation tools such as PIN [20] or Atom [21]. To identify the beginning and end of the trace, valgrind uses MPI calls (MPI call, MPI reduce, MPI send, MPI receive, etc.), although other methods can be used. Our tool runs in two different modes. In the fast forward mode, it counts the number of calls to the MPI library, introducing a 2x overhead. Then, when the cluster to trace begins, the tool executes in detailed mode, collecting data of every executed instruction and storing each cluster in a separate trace file in 64 bits.

The UPC-BSC trace format consists of three different files. First, a static basic block dictionary contains the information of all the basic blocks of the code. Each basic block description contains the basic block identifier, the address of the first instruction and the description of all its instructions, specifying the opcode, and the source and target registers. Second, we store the address of all accesses to the memory hierarchy in program order. Finally, we store the order of execution of the different basic blocks. Having a static basic block dictionary allows simulating the effect of wrong path execution.

3.6 Microarchitecture simulation

Using the instruction level trace, we can perform a number of detailed analysis studies and performance simulations using a trace-driven simulator of an out-of-order multithreaded processor.

We use MPsim simulator [22], a highly flexible cycle-accurate simulator which constitutes a multi purpose simulation tool. MPsim provides great functionality to computer architects by means of a highly-flexible parameter interface. It allows simulating a wide range of processor types, including both single core (superscalar, SMT) and multi-core (CMP, CMP+SMT) architectures. It also allows simulating homogeneous and heterogeneous configurations, providing a highly flexible parameter interface. Among the list of parameters that can be defined by the user, we have the instruction fetch policy, the branch predictor, the execution pipelines, and a fully definable memory subsystem, both in number of components and connections between them.

The core model of MPsim is an evolution of SMTsim [6], an SMT simulator for single core architectures. MPsim is a cycle-accurate simulator in which each simulated core is comprised of at least 8 pipeline stages, although the pipeline depth can be modified by adding decode or execution stages. In order to reduce computational costs, MPsim provides a trace-driven front-end. However, MPsim also permits simulating the impact of executing wrong-path instructions (when a branch miss predictions occurs) as it has a separate basic block dictionary containing the information of all static instructions of the trace. Finally, MPsim is capable of modeling ALPHA and PowerPC-like pipelines.

The MPsim memory subsystem is accurately modeled, having a complete cache hierarchy with up to three levels of caches and main memory. Bus conflicts to access shared levels of cache and main memory are modeled. All caches are multibanked and multiported, offering a wide range of configurations to the user.

Table 1. Baseline MPSim processor configuration

Architecture	2 cores, 2-way SMT, superscalar architecture
Fetch/Issue/Retire width	8/5/5 instructions per cycle
Fixed-point and load/store issue queue	36 entries
Floating point issue queue	20 entries
Branch instructions issue queue	12 entries
CR-logical instructions issue queue	10 entries
Vector instructions issue queue	36 entries
Reorder buffer	100 entries
Branch predictor	16K-entry gshare
L1 instruction cache	64KB, direct mapped, 128B line, 1 cycle hit
L1 data cache	32KB, 2-way, 128B line, 2 cycle hit, LRU
L2 unified cache	1MB, 8-way, 128B line, 15 cycle latency, LRU
Memory latency	250 cycles
Peak memory bandwidth	2 GBps per GHz

In order to detect the main bottlenecks of HPC parallel applications running in a real machine, the simulator parameters have been carefully chosen to model a PowerPC970-like processor. The main characteristics of this machine are summarized in Table1. Next, different studies have been done to find out the performance bottlenecks of these applications, including simulations with a perfect branch predictor, perfect first and second cache levels, different memory bandwidths, different number and latency of execution units, and different size of issue queues and the global completion table.

3.7 High Level Simulation

Using the original high-level trace (Section 3.1) , which describes the application behavior at the level of the programming model, and the simulation results of the individual CPU bursts (Section 3.6), which provide performance projections on a particular CPU configuration, we can rebuilt the performance of the entire application.

To perform this high level simulation we use the Dimemas simulator [12]. Dimemas reconstructs the time behavior of a parallel application on a machine modeled by a set of performance parameters. The simulator model corresponds to Figure 5. It is composed of a network of SMP nodes. Each node has a set of processors and local memory, used for communications within the node. The interconnection network is represented with two parameters: number of links from a node to the network, represented with L, and number of buses in the network, represented with B. These parameters limit network capacity, up to B messages can use concurrently the network, allowing the network contention analysis. Parameter L limits the number of messages coming in and going out for a given node, thus a connectivity analysis can also be performed. For a detailed explanation about how point-to-point and collective communications are modeled, see [23]

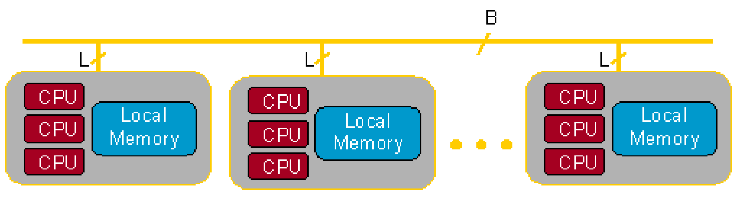


Fig. 5. Dimemas simulated model diagram

Dimemas trace To be able to execute the simulator we need to translate the Paraver trace to a Dimemas format. This trace contains the succession of resources requests made by different tasks, without timestamps. Basically, the resources request are expressed using two different records:

– Computation records. The tasks wants to perform a computation. The computation resources are expressed in time using the CPU, and are translated directly from computation states in Paraver trace. This means that the computation resources expressed in the trace correspond to the CPU consumption times made in the original application execution.
– Communication records. The task wants to perform a communication. It could be a point-to-point communication (send, receive, wait, etc.) as well as a collective communication (barrier, gather, scatter, etc.). In this case, the translation is done using a combination communication records and events of the Paraver trace.

As in Paraver trace, the are also event records. These records are not simulated in Dimemas but we use them to implement some features that modify the simulation behavior.

Baseline configuration The reference simulated architecture is detailed in Table 2. This configuration mimics the MareNostrum supercomputer design. It is a cluster of IBM JS21 server blades. Each node has two PowerPC 970MP processors (4 cores in total) and 8Gb of RAM memory. The nodes are connected using a Myrinet network. In this table, memory and network latency refer to the time added by the simulator to each communication, in terms of library initialization, not the actual latency of this units.

Table 2. Baseline Dimemas cluster configuration

Number of nodes	2560
Processors per node	4
Input links per node	1
Output links per node	1
Number of buses	∞[2]
Memory bandwidth per node	600 MB/s
Memory latency	$4\mu s$
Network bandwidth	250 MB/s
Network latency	$8\mu s$

CPU ratios. A key feature for our study is the ability of Dimemas to apply a multiplicative ratio (or *factor*) to computation requests. The default simulation of a CPU resource request consists of advancing

[2] Contention will be only defined by input and output links

the simulation clock the requested time. Remember that this requested time is the time used by the application in the traced execution. Using the CPU ratio, we can modify the CPU time request so as to simulate CPUs different to those used in the original application execution.

The most frequent use case for CPU ratios is applying just one factor to the whole execution. Using the clustering events we introduced to mark different regions in the application, we can apply different ratios to the computation regions. We call this feature *module ratios* and is the linking point between the performance projections obtained in the cycle accurate model and the Dimemas model.

4 Performance analysis

Using the instruction level trace we can analyze the characteristics of the application in order to help the user improve the performance of the application on a particular architecture. In this section, we perform several studies using instruction level traces obtained applying the methodology explained in the previous sections.

First, we describe the applications under study, and use them to show how our methodology provides orders of magnitude of reduction in the amount of simulation to be performed for the analysis. Second, we perform several studies focused on the sensitivity of the considered applications to several architectural parameters. Third, we study the performance of the applications on not (yet) available architectures. Finally, we perform a detailed study about the impact of L2 cache size on the performance of the considered applications.

4.1 Parallel HPC Applications

The message-passing parallel applications used in our experiments and presented in this paper are:

– **Versatile Advection Code (VAC)** [24]. The Versatile Advection Code is a general tool for solving hydrodynamical and magnetohydrodynamical problems arising in astrophysics. In this experiment, VAC models the propagation and evolution of interplanetary shocks. Traces are obtained for a 3D simulation from the Sun to the Earth with a grid resolution of 1200x91x180 cells. We execute VAC on 128 MPI tasks on MareNostrum supercomputer [25].
– **Weather Research and Forecasting (WRF) Model** [26]. It is a mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. In the experiments, we used the non-hydrostatic mesoscale model (WRF-NMM). The trace was extracted in an execution of 128 MPI tasks on MareNostrum supercomputer
– **NAS Parallel Benchmarks (NPB) BT** [27]. This of the well-known NAS Parallel Benchmarks that computes a finite difference solution to the 3D compressible Navier-Stokes equations. In the experiments, we used the class C variant of this benchmark, to obtain a trace of 64 MPI tasks on MareNostrum supercomputer

In table 3 we summarize the results obtained applying our methodology to these applications. Note the strong reductions achieved in the number of instructions we have to simulate compared to the full application.

Table 3. Results obtained applying our methodology to WRF, VAC and BT applications

Application	Initial Size	Reduced Size	Clusters	Representatives	Total Instructions	Reduced Instructions
WRF	5.06GB	18.1MB	5	2	$8.73 \cdot 10^{10}$	$6.46 \cdot 10^{6}$
VAC	223MB	4.2MB	2	4	$4.33 \cdot 10^{13}$	$1.48 \cdot 10^{8}$
BT	876MB	16.7MB	4	1	$3.64 \cdot 10^{11}$	$1.37 \cdot 10^{7}$

4.2 Microarchitecture bottleneck detection

The target of this experiment is to identify potential bottlenecks and give some hints on which improvements of the processor architecture would have a significant gain on the application performance. The approach has been to select a set of micro-architectural parameters to analyze their impact on the performance. The selected parameters are listed in table 4.

Table 4. Parameters used in microarchitecture bottleneck detection and their ideal values

Parameter	Description	Ideal value
`bp`	Branch predictor hit ratio	100% hit rate/1 cycle lat.
`L1I`	Size and latency of L1 instruction cache	∞ size/1 cycle lat.
`L1D`	Size and latency of L1 data cache	∞ size/1 cycle lat.
`L2`	Size and latency of L2 cache	∞ size/1 cycle lat.
`membw`	Memory bandwidth	∞
`nFU`	Number of functional units	∞
`lFU`	Latency of the functional units	1 cycle lat.
`ROB`	Size of the reorder buffer	∞ size

To determine the sensitivity of applications to different processor configurations we use an approach focusing on boundaries. The lower bound corresponds to the current platform, defined in table 1. The upper bound is an ideal situation defined in table 4. The first set of configurations presents a processor with all parameters nominal, or real, (values expressed in table 1) and one factor ideal. This set is useful to evaluate applications sensitivity to an improvement in a parameter. This set is expressed in the different tables and charts as `ideal_[parameter_name]`. On the other hand, to measure the sensitivity to the deterioration in one parameter, we use a set of configurations with all parameters ideal and one of them nominal. In this case, this set is identified in tables and charts as `real_[parameter_name]`.

In those cases where the improvement/deterioration of one parameter implies a high variation of the performance, we 'close the cycle' simulating the application in Dimemas. For these cases the target machine simulated corresponds to the baseline cluster defined in table 2.

The experiments done in this section were performed using VAC and WRF applications. We discarded BT due to its benchmark nature, so as to focus on real applications.

VAC Charts of figure 6 collect the results of the MPSim simulations for the different configurations defined per each pair of cluster and representative. In this figure, we applied a normalization so as to show the relative IPC obtained on each configuration with respect to the IPC of the ideal processor configuration. We can see that there is almost no difference behavior between the representatives chosen.

First chart, 6(a) , represents the relative IPC obtained using a real processor configuration with ideal units. In this case, we observe that the highest improvements on relative IPC (around 5%) are related to cache memory (`ideal_L1D` and `ideal_L2` configurations). In addition, ideal arithmetic units latency (`ideal_lFU`) and unlimited size of reorder buffer (`ideal_ROB`) also obtains an important improvement on IPC. Similarly, chart6(b) reflects the sensitivity of VAC to L2 cache degradation (`real_L2`), but specially to arithmetic units latency (`real_lFU`). This second chart shows us that VAC is not specially sensible to degradation of the reorder buffer.

With the considerations done previously, Table 5 shows the execution times obtained using Dimemas with the processor configurations selected. With this table, we can conclude two major facts: 1) in terms of improvement, VAC obtains the better execution time when using an ideal L1 data cache, an ideal L2 cache or an ideal arithmetic units latency. This suggests that the programmer two things: he should try to redesign the data access, in order to improve the cache use, and also try to improve the instruction

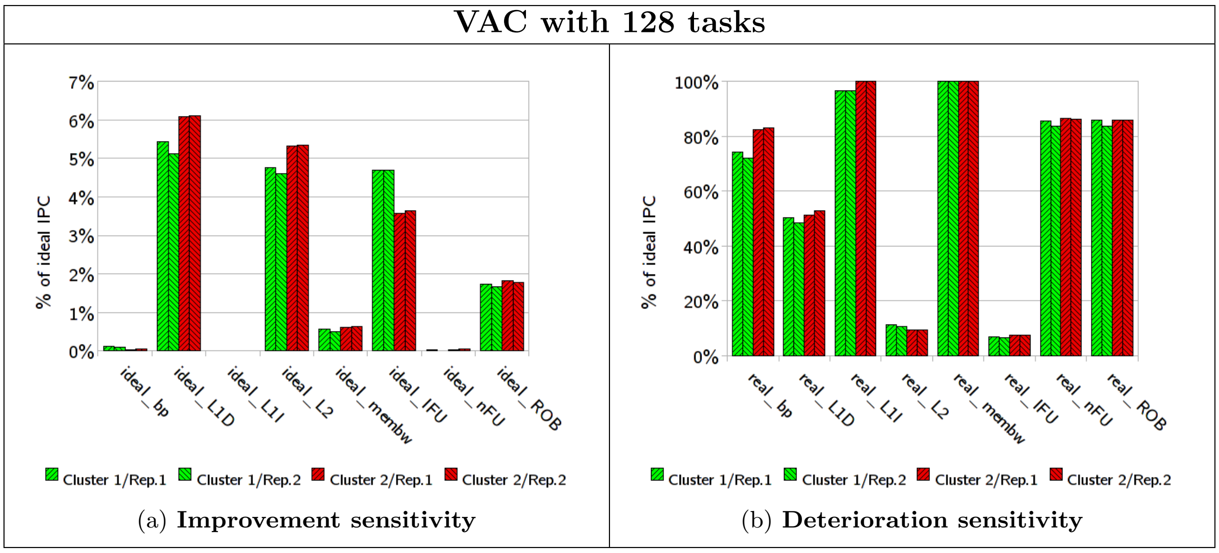


Fig. 6. VAC application relative performance with respect to ideal IPC using different CPU configurations

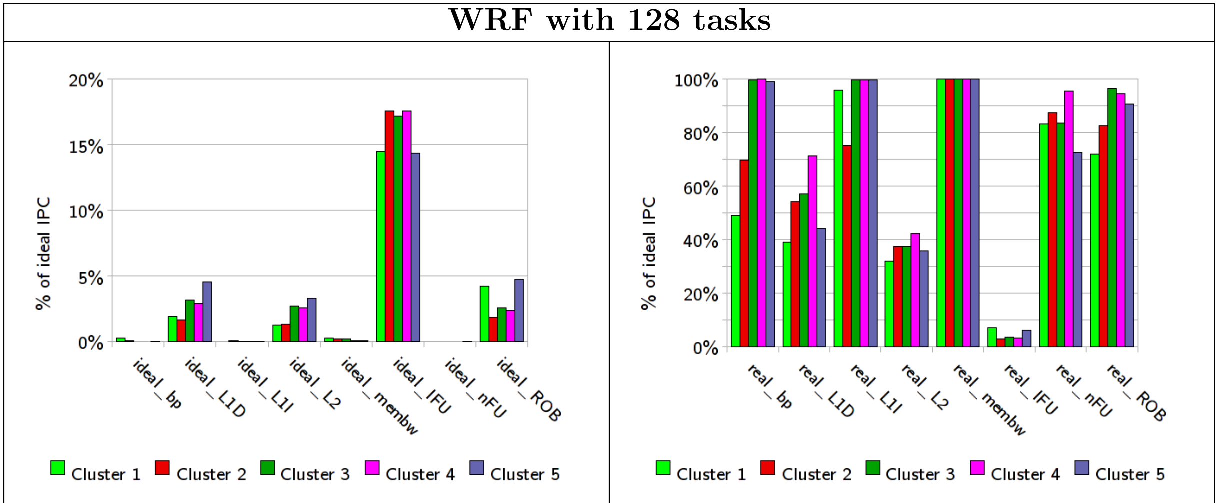
dependency chain; 2) in terms of deterioration, VAC is sensible to arithmetic units latency (45.57% of ideal IPC when using `real_lFU` configuration), so that confirms the previous indications. Similarly, in terms of hardware design, we can affirm that VAC would benefit of using a bigger cache. This affirmation is deeply related with the previous experiment, point 4.4 , regarding to cache sizes, which indicates us that a 32MB L2 cache obtains the target IPC.

Table 5. Predicted execution times of interesting processor configurations. VAC application with 128 tasks

Configuration	IPC Cluster 1	IPC Cluster 2	Avg. IPC	**Exec. Time**
`ideal`	4.496	4.266	4.381	**0.930**
`real_L2`	0.689	0.659	0.674	**4.628**
`real_lFU`	0.689	0.659	0.674	**5.515**
`ideal_L1D`	0.629	0.609	0.619	**6.021**
`ideal_L2`	0.629	0.609	0.619	**6.271**
`ideal_lFU`	0.605	0.512	0.558	**6.577**
`ideal_ROB`	0.483	0.441	0.462	**7.959**
`real`	0.414	0.371	0.392	**9.354**

WRF As in previous experiment, charts of Figure 7 summarize the percentage of ideal IPC obtained using different processor configurations.

WRF is sensitive to the improvement (chart 7(a)) of L1 data cache (`ideal_L1D`), L2 cache (`ideal_L2`), arithmetic units latency (`ideal_lFU`) and reorder buffer size (`ideal_ROB`). However, in this application we observe a higher IPC improvement, around 16%, when using ideal arithmetic units. On the other hand, the deterioration sensitivity of WRF (chart 7(b)) is quite similar to VAC, because both are sensitive to L1 data cache (`read_L1D`), L2 cache (`real_L2`) and arithmetic units latency (`real_lFU`) and not to reorder buffer size (`real_ROB`), as we can expect looking at the improvement observed in previous configuration set. The main difference in this case is the degradation achieved using a real L2 cache, as defined in table 1, is not that bad as the observed in VAC.



(a) **Improvement sensitivity** (b) **Deterioration sensitivity**

Fig. 7. WRF application relative performance with respect to ideal IPC using different CPU configurations

Table 6 present the average IPC per cluster, the total IPC average of all clusters and finally the Dimemas predicted execution times for the configurations with bigger improvements/degradations. In this case, it is interesting to compare the average IPC obtained on each configuration with the predicted execution time. There are two interesting cases: 1) just using the real latency of arithmetic units (`real_lFU`) configuration, the average IPC degrades to 4.66% of the ideal IPC, and execution time becomes 5.25 times slower than ideal; 2) on the other hand, with this same metric, but quantifying the improvement, when using a real processor with ideal arithmetic logic latencies (`ideal_lFU`), the average IPC is just 16.14% of ideal IPC, but the execution time is 2.18 times faster than the real execution. With this two hints, we conclude that WRF is extremely sensitive to the latency of arithmetic units. This can be interpreted as the dependency chain between instructions is hard. Having this information, a good advice to the developers should be redesign the code to avoid this dependencies (e.g., unrolling loops).

Table 6. Predicted execution times of interesting processor configurations. WRF application with 128 tasks

Configuration	IPC Cluster 1	IPC Cluster 2	IPC Cluster 3	IPC Cluster 4	IPC Cluster 5	Avg. IPC	**Exec. Time (s)**
`ideal`	5.229	4.140	3.433	3.889	3.906	4.119	**0.120**
`real_L1D`	2.437	2.446	2.153	2.918	2.041	2.399	**0.200**
`real_L2`	2.118	1.832	1.556	1.929	1.757	1.838	**0.240**
`real_lFU`	0.969	0.549	0.542	0.589	0.758	0.681	**0.630**
`ideal_L1D`	0.735	0.504	0.534	0.581	0.708	0.612	**0.730**
`ideal_L2`	0.705	0.491	0.521	0.570	0.667	0.591	**0.760**
`ideal_lFU`	1.312	1.093	0.953	1.081	1.038	1.095	**0.380**
`ideal_ROB`	0.842	0.511	0.516	0.565	0.716	0.630	**0.690**
`real`	0.647	0.443	0.438	0.482	0.556	0.513	**0.830**

4.3 Multi-core study

The aim of this experiment is to predict the application performance using a different machine configuration. As said in the introduction, one of the strongest points of our methodology is the ability to simulate non-existent computers in a fast and easy way. We can predict the performance varying a wide range of parameters of the target architecture: from microarchitectural design (functional units, pipelines, etc.) to different network characteristics (network bandwidth, number of links per node, etc.).

With this experiment we show the impact of using multi-core chips in the nodes. In this case, we used the WRF application with 128 tasks trace. We modify the CPU used in the single-processor nodes, increasing the number of cores per processor. These cores are defined by the parameter of table 1, but we tuned the cache size, the number of cache banks and the number of ways to reflect a realistic scaling in the architecture. Table 7 resumes the different processor combinations of the target system, as well as the cache configurations per combination. As a note, considering that we chose 2 representatives per cluster, when we simulated chips with more than 2 cores, we replicated the representatives in MPSim adding a skew to memory access patterns.

In terms of network capabilities, table 8 shows the parameters used in Dimemas simulations. We chose different network bandwidths and different number of input/output links to evaluate the trade-off between improving the network or just adding more networks cards of this unreal computer. In all cases, the number of buses is considered infinite.

Table 7. Node/processor configurations used in the multi-core experiment

# of nodes	Cores per processor	Cache size	Cache Banks	# of Cache ways
128	1	1MB	4	4
64	2	2MB	8	8
32	4	4MB	8	8
16	8	8MB	16	16
8	16	16MB	16	16

Table 8. Network configurations used in the multi-core experiment

Configuration	Memory BW.	Network BW.	Input links	Output links	Memory latency	Network latency
`IDEAL`	∞	∞	∞	∞	0	0
`5GBps_M_250MBps_N_1L`	5GB/s	250MB/s	1	1	$4\mu s$	$8\mu s$
`5GBps_M_250MBps_N_2L`	5GB/s	250MB/s	2	2	$4\mu s$	$8\mu s$
`5GBps_M_250MBps_N_4L`	5GB/s	250MB/s	4	4	$4\mu s$	$8\mu s$
`5GBps_M_250MBps_N_8L`	5GB/s	250MB/s	8	8	$4\mu s$	$8\mu s$
`5GBps_M_500MBps_N_1L`	5GB/s	500MB/s	1	1	$4\mu s$	$8\mu s$
`5GBps_M_1GBps_N_1L`	5GB/s	1GB/s	1	1	$4\mu s$	$8\mu s$

With this experiment design, we expected to see two effects in the application performance. First, the average IPC should degrade: the resource sharing in a multi-core processor implies a worst average performance than having a replication of full hardware. Second, the communication performance should be improved: having more tasks in the same node implies that if these tasks communicate each other, these communications will be faster.

Table 9 confirms our first statement. In this table we can see how average IPC of different clusters degrades while increasing the number of cores per processor. This degradation affects each cluster in a different way: clusters 1, 2 and 3 IPCs are steady up to 4 cores per processor, while the 8-core penalty for clusters 4 and 5 is still acceptable. This point us to recommend not to use processors with more than 4 cores for this application unless higher memory bandwidth can also be provided.

The second statement is confirmed by the information shown in chart 8(b). In this chart we can see how different communication times tend to decrease when more cores are added. In this chart we can observe that the configurations with one input and one output link (`*_1L`) an intermediate zone with higher communication times appears. This is caused by the contention produced in the nodes due to the simultaneous inter-node communications. This problem is solved adding more links per node.

As a final note, looking at chart 8(a), it is interesting to answer the following question: why if we reduce the communication time, the execution time increases? It is clear that communications times decrease when we increase the bandwidth or we add more links per node, as can be seen in 8(b). However, execution times seem steady for all configurations. The key point is follow the `IDEAL` execution time. We can see that all other configuration is nearly the same. This fact shows us that WRF the computation time is the main factor in the total execution time. The configuration used with 128 MPI tasks produces a good data distribution, and the improvement of the communications is not balanced with the deterioration of computation. Our conclusion for this question is: WRF is a pretty well optimized code.

Table 9. WRF resulting average IPC per cluster in different multi-core processor configurations

Processor cores	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
1	0.653	0.472	0.439	0.479	0.552
2	0.644	0.441	0.434	0.479	0.554
4	0.632	0.431	0.417	0.474	0.550
8	0.596	0.408	0.376	0.473	0.540
16	0.514	0.335	0.255	0.407	0.374

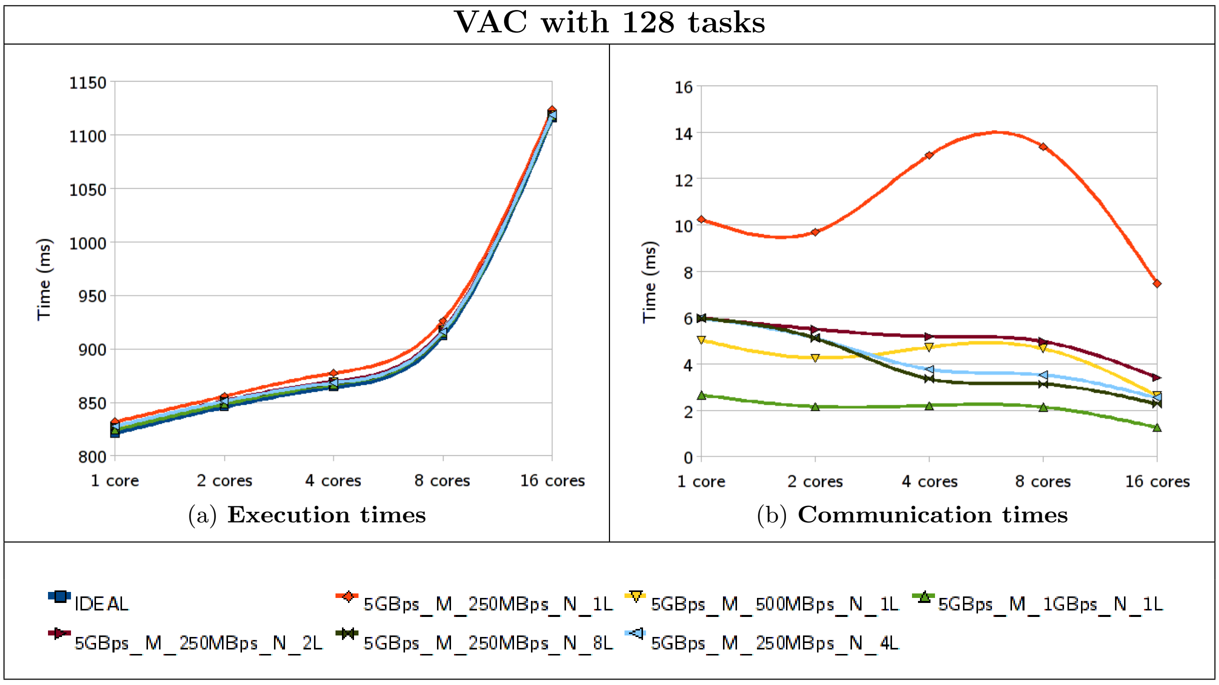


Fig. 8. WRF execution times and communication times for different network configurations using multi-core nodes

4.4 Trade-offs between L2 cache size and network bandwidth in application performance

The aim of the experiments reported in this section is to study in detail the evolution of the behavior of applications as the size of L2 cache and the network bandwidth are increased. This study can provide valuable insights to take critical decisions about the architecture of high performance computing infrastructures, to evaluate the adequateness of a specific propose HPC machine or to select the more suitable application to execute on a specific machine.

In figure 9(a) we show results extracted from VAC application in terms of IPC. Remember that, in this case, we have divided the computing time in two clusters. From the point of view of the size of L2 cache, we can see the same behavior in these two clusters, that is, a remarkable increase of the IPC of the application if the size of the L2 cache is higher than 2 MB. The most important conclusion we can extract from this fact is that we can improve strongly the performance of this application if we execute it on a supercomputing infrastructure whose processors have a L3 cache level, able to reduce the cycles dedicated to access principal memory due to L2 misses.

In figure 9(c) we show the experiments performed taking into account NAS-BT class C application in terms of IPC. In the experiments we observe a strong sensitivity of Cluster 2 as we increase L2 cache size. According to the bibliography [27], the problem size of NAS BT class C is equal to 4.25 MB. In the results obtained, we can confirm this fact taking into account the stabilization of performance improvement if the L2 cache size is higher than 4.2 MB.

In figure 9(e) we depict the same results in case of WRF application. In this case, we can see a strong improvement of the performance in all the clusters until 1 MB of L2 cache size is reached. This behavior is due to the size of the input data.

In figures 9(b), 9(d) and 9(f) we show the execution times obtained using the IPC values shown above to approximate the computation times and Dimemas to simulate communications. This approach enables us to study at the same time the impact of architectural parameters, in this case L2 cache size, and network parameters, in this case bandwidth. We can divide applications in two sets, according to the results obtained. On the one hand, we have VAC and WRF applications. In these cases, the impact of the network is negligible and the dominant performance factor is IPC. On the other hand, we have BT application. In this case, the impact of the network bandwidth over the execution time is significant. We have that a reduction of L2 cache size from 2MB to 64KB can be nearly compensated by an increase on the communication bandwidth from 250Mb/s to 500Mb/s. This is a significant trade-off between architecture and network that our approaches enables to evaluate numerically thanks to its multi-scale simulation capabilities.

5 Conclusions

In this paper we have described the simulation methodology used at the Barcelona Supercomputing Center. This methodology allows very detailed performance analysis and performance prediction for full-scale MPI applications running on hundreds of processors in reasonable time. Compared to previous approaches, the main advantage of our approach is that it can provide results not only across well known systems, but also for novel architecture features not yet available in the market.

The method is based on the performance isolation provided by the MPI programming model and the distributed memory cluster architecture: the computational performance of one MPI task does not depend on what happens on the other parallel tasks. The same isolation property can be observed on task-based programming models and DMA-based architectures such as StarSs on the CellBE.

Shared memory programming models, like OpenMP, or a mixed MPI+OpenMP application would require that we simulate in cycle-accurate mode all the CPU bursts executing on the shared memory node in order to account for the impact of the cache coherency protocol, limiting our method to cluster architectures with nodes featuring only a few shared memory processors.

The most immediate future work includes extending this methodology to other programming models that maintain the task independence assumption, like StarSs. Further research is also ongoing to extend this methodology to large-scale shared memory systems.

References

1. Casas, M., Badia, R.M., Labarta, J.: Automatic structure extraction from MPI applications tracefiles. In: Euro-Par. (2007)
2. Rosenblum, M., Herrod, S.A., Witchel, E., Gupta, A.: Complete computer system simulation: The simos approach. IEEE Parallel and Distributed Technology **3** (1995) 34–43
3. Bellard, F.: Qemu, a fast and portable dynamic translator. (2005) 41–46
4. Bedichek, R.: Simnow: Fast platform simulation purely in software. (Aug 2004)
5. Austin, T., Larson, E., Ernst, D.: Simplescalar: an infrastructure for computer system modeling. Computer **35**(2) (Feb 2002) 59–67
6. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism. In: ISCA. (1995)
7. Moudgill, M., Wellman, J.D., Moreno, J.H.: Environment for powerpc microarchitecture exploration. IEEE Micro **19**(3) (1999) 15–25
8. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35**(2) (Feb 2002) 50–58
9. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., Ortega, D.: Cotson: infrastructure for full system simulation. SIGOPS Oper. Syst. Rev. **43**(1) (2009) 52–61
10. Laura Carrington, Allan Snavely, X.G.N.W.: A performance prediction framework for scientific applications. In: Proceedings of the International Conference on Computational Science (ICCS). (2003) 926–935
11. San Diego Supercomputing Center - PMaC laboratory: Metasim. http://www.sdsc.edu/pmac/MetaSim/Tracer/tracer.html
12. BSC Performance Tools Team: Dimemas Home@BSC. http://www.bsc.es/plantillaA.php?cat_id=475
13. BSC Performance Tools Team: Paraver and Tracing toolkit Home@BSC. http://www.bsc.es/plantillaA.php?cat_id=485
14. Timothy Sherwood, Erez Perelman, G.H., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS. (2002)
15. Casas, M., Badia, R.M., Labarta, J.: Automatic phase detection of mpi applications. In: Parco. (2007)
16. Gonzalez, J., Gimenez, J., Labarta, J.: Automatic detection of parallel applications computation phases. In: IPDPS. (2009)
17. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.K.: Measuring benchmark similarity using inherent program characteristics. IEEE Transactions on Computers **55**(6) (2006) 769–782
18. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Second International Conference on Knowledge Discovery and Data Mining, Portland, Oregon, AAAI Press (1996) 226–231
19. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI. (2007)
20. keung Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Janapa, V., Hazelwood, R.K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI, ACM Press (2005) 190–200
21. Srivastava, A., Eustace, A.: Atom - a system for building customized program analysis tools. In: PLDI. (1994) 196–205
22. Acosta, C., Cazorla, F.J., Ramirez, A., Valero, M.: The mpsim simulation tool, Technical Report UPC-DAC-RR-2009-7 (january 2009)
23. Girona, S., Labarta, J., Badia, R.M.: Validation of dimemas communication model for mpi collective operations. In: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, Springer-Verlag (2000) 39–46
24. Toht, G.: Versatile Advection Code Homepage. http://www.phys.uu.nl/ toth/
25. (BSC-CNS), B.S.C.: Marenostrum supercomputer architecture description. http://www.bsc.es/plantillaA.php?cat_id=200
26. Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., Wang, W. In: ECMWF. (2004)
27. Bailey, D., Harris, T., Saphir, W., Wijngaart, R.V.D., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Technical report, NAS-95-020 (1995)

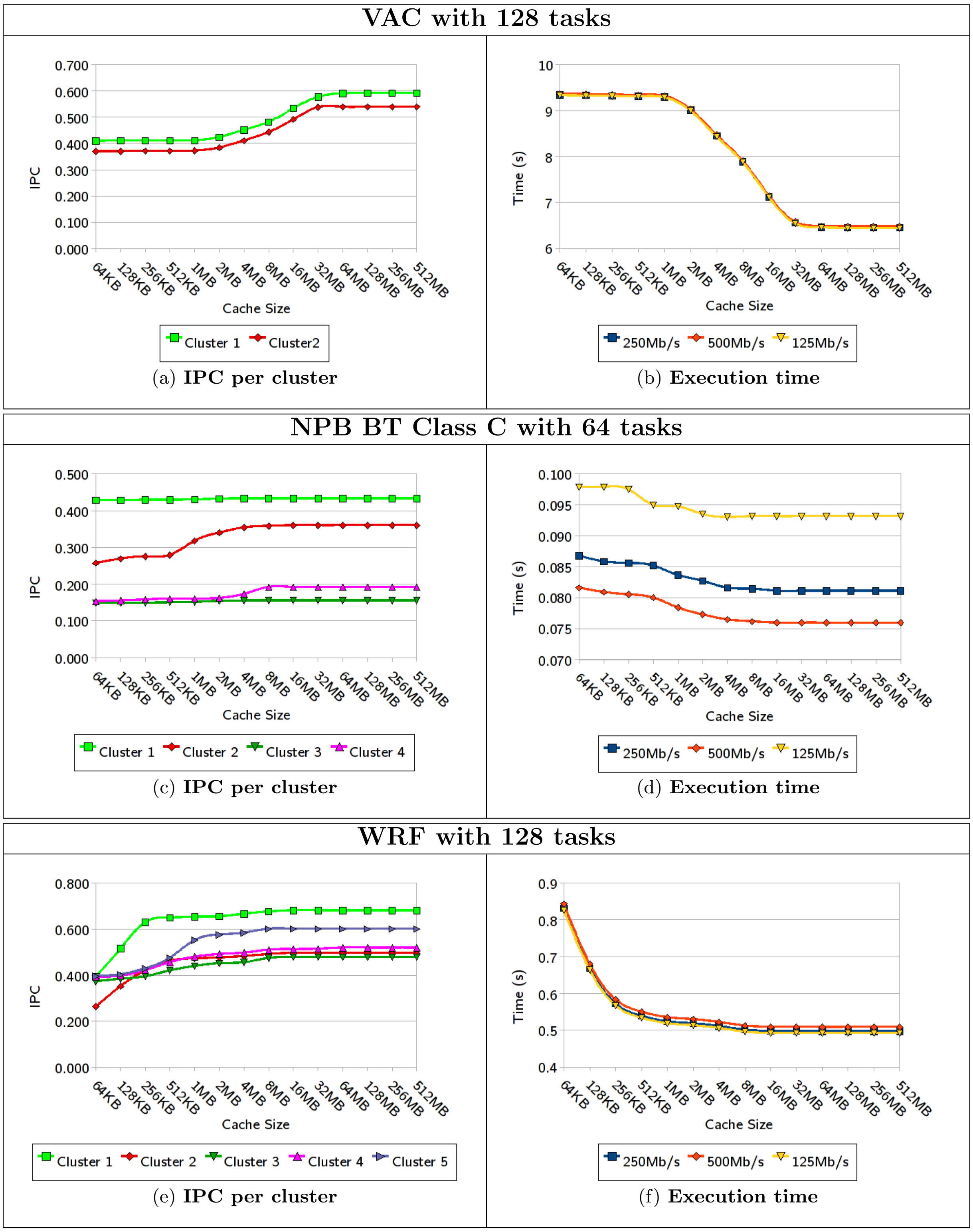


Fig. 9. IPC and execution times obtained in VAC, BT and WRF with different cache sizes and network bandwidth