SIMULATING WHOLE SUPERCOMPUTER APPLICATIONS

DETAILED SIMULATIONS OF LARGE SCALE MESSAGE-PASSING INTERFACE PARALLEL APPLICATIONS ARE EXTREMELY TIME CONSUMING AND RESOURCE INTENSIVE. A NEW METHODOLOGY THAT COMBINES SIGNAL PROCESSING AND DATA MINING TECHNIQUES PLUS A MULTILEVEL SIMULATION REDUCES THE SIMULATED DATA BY VARIOUS ORDERS OF MAGNITUDE. THIS REDUCTION MAKES POSSIBLE DETAILED SOFTWARE PERFORMANCE ANALYSIS AND ACCURATE PERFORMANCE PREDICTIONS IN A REASONABLE TIME.

Juan Gonzalez
Barcelona Supercomputing Center

Marc Casas
Lawrence Livermore National Laboratory

Judit Gimenez
Miquel Moreto
Alex Ramirez
Jesus Labarta
Mateo Valero
Polytechnic University of Catalonia

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

Architecture simulation is still sequential, so it depends heavily on the processor's single-thread performance. The advent of multicores has signaled the end of individual processor speedups and has not benefited simulator performance.

Speeding up simulation by parallelizing the simulator[1-3] is complex, and using FPGA results in specialized simulators.[4] A better alternative is to reduce the amount of simulation performed without losing relevance and accuracy. The usual approach to reduce the amount of simulation is sampling.[5,6] That is, the simulator proceeds through the application quickly without actually modeling the hardware, and at certain intervals (random, or carefully selected), it changes to a highly detailed (and slow) simulation mode. Statistic analysis of this sampling approach shows that high accuracy can be achieved through simulation of small samples. Sampling has been widely used in the past to simulate sequential applications running on a single processor. (For information on other approaches to simulation, see the "Related Work in Simulation Tools" sidebar.)

However, these sampling methodologies present several uncertain aspects when simulating parallel systems and applications: how should the application behave with regard to timing when it's 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?

Methodology

We used a fine-level characterization of the application computation, based on signal processing and data-mining techniques. Then, we selected what was simulated to minimize the time spent modeling the multiprocessor system's most time-consuming

Published by the IEEE Computer Society

Related Work in Simulation Tools

Researchers have widely used simulation tools to verify, analyze, and improve computer systems. Simulation is used at different levels of detail, depending on the particular target system to study. The trade-off between simulation speed and accuracy is always present in these studies. In this area, no other current simulation infrastructure allows the simulation of large-scale computing systems such as supercomputers at the same level of detail that our methodology provides, without compromising the total simulation time.

Functional simulators emulate the target system's behavior, including the operating system and the different system devices (such as memory, network interfaces, and disks). These simulators let designers verify system correctness and develop software before the system has been built, but they can't estimate the real system performance with the simulators. Some examples are SimOS, QEMU, and SimNow.

Microarchitecture simulators model in detail the processor's architecture and can estimate the performance of an application with different processor configurations. However, these kinds of simulators normally don't model the interaction between the architecture and the operating system and other system devices, and they tend to be expensive in terms of time. Researchers have used three major approaches to tackle this problem. The first approach is using statistical sampling to reduce the volume of instructions to simulate, as SIMPoint[1] and SMARTS[2] do. The second approach is working on the parallelization of the simulator itself, such as in Proteus[3] or the more recent Graphite.[4] The third approach is using FPGAs to implement the simulator itself (such as in RAMP Blue[5]), reducing the simulation times but also limiting the flexibility of the microarchitecture to simulate.

Full system simulators, such as SimICS and COTSon[6], include the features of functional and microarchitecture simulators at the cost of simulation time. COTSon can model a cluster supercomputer from the microarchitecture up to the operating system, but it's clearly oriented to hardware design, whereas our work focuses on performance analysis of parallel applications.

Other authors have proposed simulation methodologies to evaluate the performance of large-scale parallel applications.[7-9] Carrington et al. use the same network simulator that we do, but the microarchitecture simulation is based on signatures of all computation regions.[7] León et al. present a parallel network simulator combined with a regular microarchitecture simulator.[8] They obtain good simulation speed-ups, but the simulator parallelization adds new problems such as the high variability in the accuracy of the results across different runs. Neither approach uses any information reduction process to reduce the simulation time, resulting in time-consuming simulations.[7,8] Finally, Zheng et al. focus on selecting the computation regions that drive the application execution,[9] as we also do. However, they require the intervention of an expert (that is, the application developer) to describe the most important computation bursts, while in our project, that part is automated.

References

1. E. Perelman et al., ''Using SimPoint for Accurate and Efficient Simulation,'' *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* ACM Press, 2003, pp. 318-319.
2. R.E. Wunderlich et al., ''SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,'' *ACM SIGARCH Computer Architecture News,* vol. 31, no. 2, 2003, pp. 84-97.
3. E.A. Brewer et al., ''PROTEUS: A High-Performance Parallel-Architecture Simulator,'' *Proc. ACM SIGMETRICS Joint Int'l Conf. Measurement and Modeling of Computer Systems,* ACM Press, 1992, pp. 247-248.
4. J.E. Miller et al., ''Graphite: A Distributed Parallel Simulator for Multicores,'' *Proc. 16th IEEE Int'l Symp. High Performance Computer Architecture,* IEEE Press, 2010, doi:10.1109/HPCA.2010.5416635.
5. E.S. Chung et al., ''A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations using FPGAs,'' *Proc. 16th Int'l ACM/SIGDA Symp. Field Programmable Gate Arrays,* ACM Press, 2008, pp. 77-86.
6. E. Argollo et al., ''COTSon: Infrastructure for Full System Simulation,'' *ACM SIGOPS Operating Systems Review,* vol. 43, no. 1, 2009, pp. 52-61.
7. L. Carrington et al., ''A Performance Prediction Framework for Scientific Applications,'' *Proc. Int'l Conf. Computational Science,* LNCS 2659, Springer, 2003, pp. 926-935.
8. E.A. León et al., ''Instruction-Level Simulation of a Cluster at Scale,'' *Proc. Conf. High Performance Computing Networking, Storage and Analysis,* ACM Press, 2009, doi:10.1145/1654059.1654063.
9. G. Zheng et al., ''Simulating Large Scale Parallel Applications Using Statistical Models for Sequential Execution Blocks,'' *Proc. IEEE 16th Int'l Conf. Parallel and Distributed Systems,* IEEE Press, 2010, pp. 221-228.

components: the processors and the cache hierarchy. To do that, we reduced the number of CPU bursts (the regions between two communications in a parallel application) that had to be simulated by selecting those that better represented the whole application execution. After this precise selection, the representative set of CPU bursts was a very low number of the total present in the application.

The methodology for accurately selecting the representative CPU bursts of a parallel

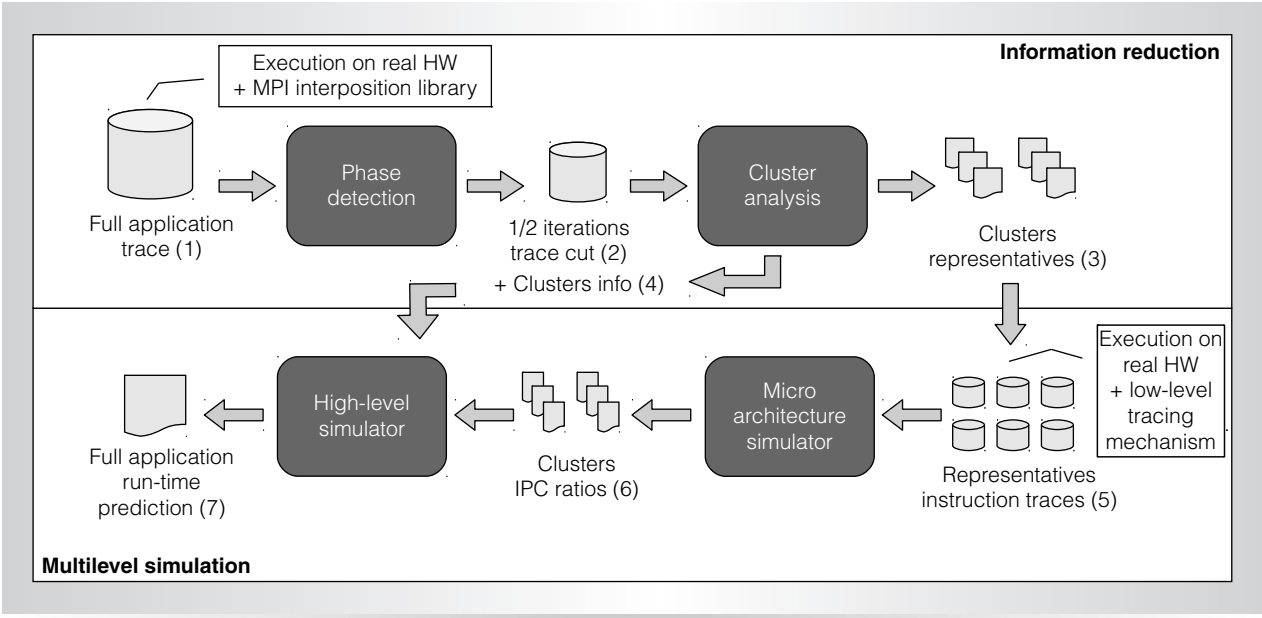

Figure 1. Simulation methodology cycle for a whole supercomputing application. Starting with a trace of a parallel application (step 1), we produce a subtrace (or trace cut) containing information of just two iterations (step 2). A cluster analysis is applied to the information of the computation regions present on this reduced trace, and a set of representatives per cluster is selected (step 3), adding cluster information to the trace cut (step 4). The set of representatives is traced (step 5) and simulated using a low-level simulator to obtain the ratios on other possible processor configurations (step 6). Finally, using a full-system-scale simulator, we combine the communication information present in step 3 and the cluster instructions per cycle (IPC) ratios (step 6) to predict the total runtime of the whole application (step 7). (HW: hardware; MPI: message-passing interface.)

application is one of our main contributions. The top of Figure 1 depicts this *information reduction* process. It is decomposed into a phase detection analysis, which can distinguish the iterations present in the parallel applications we work with, and a cluster analysis, which groups the CPU bursts of an iteration that behave similarly. Finally, the representative CPU bursts of each cluster are selected.

Then, the selected CPU bursts are simulated at the microarchitecture level to obtain their performance in the target machine to study. This information is provided to a high-level application simulator that predicts the whole parallel application's execution time. This combination of two simulators with different abstraction levels composes the *multilevel simulation* process, depicted at the bottom of Figure 1.

The combination of the information reduction process with the multilevel simulation process permits software performance analysis beyond what current performance counters would allow. Furthermore, it lets us predict the performance of an application running on a future system for which no performance data can be used as a reference machine. All these performance analyses can be done while maintaining high accuracy in performance predictions and without needing exhaustive simulations.

Information reduction

As a whole, the information reduction process focuses on describing the CPU bursts of an application using the minimum data. To start this process, we run the whole application we want to analyze to obtain an application-level trace (see Figure 1, step 1). This trace is a sequence of time-stamped records defining each task's computation and communication. The trace contains state records, which represent regions with the same semantics (such as running, communicating, or I/O), and events, which are

punctual data that enrich the state records' information (such as performance hardware counter values and call stack information). For information about this trace, the tools to obtain it, and a visualizer, see the Barcelona Supercomputing Center performance tools website (http://www.bsc.es/paraver).

Phase detection

The underlying idea of phase detection is that we can benefit from the repetitive nature of high-performance computing (HPC) applications and select a representative segment of the whole application run, containing one or more iterations of the main loop. In this article, we work with complete iterations of the main loop, but we can also apply the same kind of analysis to finer-grained levels.

Our approach is based on signals: we express an application's behavior in terms of time-varying functions generated from the hardware counter values present in the trace. Once we have obtained the signal, we apply to it the discrete wavelet transform (DWT), which has two interesting properties:

- It can be computed in $O(n)$ operations, where n is the input signal's length in terms of sampled points.
- It captures not only the values of the input signal's frequencies but also the physical location where those frequencies occur.

These two properties let us find the locations of the execution phases within the signal's domain and an approximate value of the main frequencies in each periodic execution phase.

Once we've detected execution regions with high-frequency behavior, we perform an analysis of the exact value of the main frequency within regions with high frequencies. To do so, we apply an autocorrelation function to the signal to detect.

If the execution has the typical HPC application structure, this analysis will detect the periodic phase because it has a strong high-frequency behavior. Besides, if the execution contains multiple periodic phases, the automatic system will also detect them because DWT can separate the periodic phases characterized with different frequencies.

Assuming each periodic region detected by the DWT has an iterative pattern of T seconds (of course, this value can be different for each periodic region) and since there are no significant differences between the repetitions of the pattern, it is possible to select several such regions, notably simplifying the subsequent analysis in terms of the amount of data that must be analyzed.

At the end of this step, for each periodic region detected, we generate a subtrace (or *trace cut*), which is a portion of the original trace, containing information of n iterations. Typically, using just one or two iterations (Figure 1, step 2) is enough to keep the structure of the application clearly. In addition, this analysis also gives us the *cut factor*, so as to approximate the application's total runtime by multiplying it per the subtrace duration. We use this cut factor in the final step of our methodology. (For further information on phase detection, see previous work by Casas et al.[7])

Cluster analysis

After selecting one or two iterations of the application, we must characterize the different CPU bursts present in these iterations. In this step of the information reduction process, we aim to identify the clusters of CPU bursts that cover most of the application's computation phases. To detect similarities between CPU bursts, we use hardware counter data available on the real machine. Then, we select a set of representatives per cluster.

Each CPU burst is presented in the trace with a large set of metrics describing its performance (duration plus up to eight hardware counters). We select a subset of the counters that will be used as the parameters for the cluster analysis algorithm. We've obtained good results for most cases when using completed instructions and instructions per cycle (IPC) to characterize CPU bursts. These performance counters focus on a general performance view of the application and can detect regions with different computational complexity (thanks to the instructions-completed counter), while simultaneously differentiating between regions

with the same complexity but different performance (thanks to the IPC metric).

We used the Density Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm in our study.[8] The algorithm has two parameters: Epsilon defines the range of neighborhood queries to find groups of closer points, and MinPoints defines the minimum number of points present in such a neighborhood to consider it a cluster. The latter parameter is required to filter nonrepresentative clusters of CPU bursts (which the algorithm identifies as noise). The main point is that this algorithm doesn't assume any distribution of the data to cluster. That is especially important because, as we have observed, the performance hardware counter data doesn't have a consistent underlying distribution.

The results of applying the computation bursts' characterization show us the structural similarities of computation bursts among all application tasks—that is, the application's low-level computation structure. This information is added to the trace cut, as a pair of events, wrapping each computation burst (Figure 1, step 4). In fact, this method's underlying idea is similar to phase detection, but at a finer granularity level. (For more information about cluster analysis, please refer to previous work by Ester et al.[8] and Gonzalez et al.[9])

Cluster representatives selection

After the CPU bursts characterization, we select a reduced number of representatives from those clusters that represent a significant percentage of the application's execution time. Only these cluster representatives will be simulated at the microarchitecture level. We consider the minimum number of clusters that cover more than 80 percent of the total execution time of the application—usually less than six clusters.

Selecting the cluster representatives implies two decisions: first, select a small subset of tasks (from 1 to 5) where representatives will be taken, and second, select the CPU bursts themselves to be traced. Our experiments show that there is no significant difference between the selection schemes, so we chose the representatives at random.

This selection results in a reduced set of CPU bursts (no more than 10 in our validation experiments) that precisely represent the different computation behaviors in the application trace. Furthermore, we also must know the exact location of these CPU bursts in the application execution (Figure 1, step 3) in order to obtain the instructions trace needed to start the second part of the methodology, the multilevel simulation process.

Multilevel simulation

The parallel application's simulation process comprises two steps with different levels of detail. Once the cluster representatives have been selected, we proceed with the microarchitectural simulation (Figure 1, step 4). These simulations let us predict the behavior of all the computation regions in the target machine we plan to evaluate. These results are provided to the application-level simulator (Figure 1, step 5), which estimates the whole application's execution time (Figure 1, step 6) using the cut factor obtained before (Figure 1, step 2).

Microarchitecture simulation

To obtain a microarchitecture trace at the instruction level, we must rerun the application with a low-level tracing mechanism. This trace describes in detail the source and target operands of each instruction, the instruction code, and the addresses of the memory accesses. To obtain this trace, we use valgrind. To identify where the different cluster representatives begin and end, we count the number of message-passing interface (MPI) calls performed before the selected CPU burst begins.

We use MPsim, a cycle-accurate simulator in which each simulated core comprises at least eight pipeline stages, although we can modify the pipeline depth by adding decode or execution stages. To reduce computational costs, MPsim provides a trace-driven front end. However, it also supports 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.

The MPsim memory subsystem is accurately modeled, having a complete cache hierarchy with up to three levels of caches

and main memory. The simulator also models bus conflicts to access shared levels of cache and main memory. All caches are multibanked and multiported, offering a range of configurations to the user. MPsim optimistically assumes that main memory is perfect and, thus, all memory accesses will hit.

Previous performance studies using phase detection analysis[7] and cluster analysis[9] were designed to perform only high-level simulations of the application. In these cases, the original machine's performance counters provide the performance information. Adding a microarchitecture simulator to the simulation methodology lets us predict different target machines' performance, opening a range of new possible performance studies.

Application simulation

Using the application trace cut produced after phase detection with the clusters information (Figure 1, step 4) and the performance predictions per cluster (Figure 1, step 6), we can rebuild the entire application's performance in the target machine to study.

To perform this high-level simulation, we used the Dimemas simulator.[10] Dimemas reconstructs the time behavior of a parallel application on a machine modeled by performance parameters. The simulator model comprises a network of nodes, each containing a set of processors and local memory, used for communications within the node. The model's main parameters include the memory latency and bandwidth to describe the local communications inside a node, the network latency and bandwidth to describe the communications using the cluster network, and the total number of concurrent messages on the network to describe the contention.

CPU ratios. A key feature for our study is the ability of Dimemas to apply a multiplicative ratio to computation requests. A CPU burst's default simulation consists of advancing the simulation clock by the length of the CPU burst itself. Remember that the application used this time in the real execution. Using the CPU ratio, we can modify the CPU time request to simulate different CPUs. Applying different ratios to the clusters found using the timings produced by the microarchitectural simulation, we can accurately predict the timings of the application's computation parts. Because we select only those clusters that cover 80 percent of the computation time, we have the ratio for just a subset of all CPU bursts. To consider those CPU bursts that don't belong to the main clusters, we apply to them the weighted average of the ratios we actually computed.

Full application runtime projection. Once we've simulated the iterations present on the trace cut using the CPU ratios, we have the runtime prediction for this segment of the trace. To obtain the full application runtime, we multiply this runtime by the cut factor we obtained during phase detection.

Experimental validation

The main purpose of experimental validation is to quantify which errors are introduced in each step of the methodology. Because it's divided into two processes, we distinguish two types of potential errors:

- *representativity errors*, or evaluations of the quality of the representatives obtained in the information reduction process, with respect to the information present on the original trace; and
- *simulation error*s, or the errors introduced by the two different simulators.

Finally, we must also consider the errors resulting from applying the whole methodology along with the reduction in simulation time from applying our simulation methodology.

We conducted the validation experiment using two applications at production in our supercomputing facility: Versatile Advection Code (VAC)[11] and the Weather Research Forecasting (WRF) model.[12] In both cases, the applications executed with 128 tasks on the MareNostrum supercomputer. In the case of WRF, initialization and finalization phases covered a significant part of the application's execution time. This behavior was detected with our simulation methodology, which suggests that we should focus only

Table 1. Reduction factors and quality evaluation of the information reduction process parts.

	Weather Research Forecasting (WRF)				Versatile Advection Code (VAC)			
Parameter	All trace	Two-iteration cut	Clusters	Representatives	All trace	Two-iteration cut	Clusters	Representatives
No. of bursts	4,137,194	51,008	1,274	10	523,616	10,495	746	8
Reduction factor	—	81.1	3,247	413,719	—	49.9	702	65,452
No. of instructions	8.29×10^{12}	1.0×10^{11}	8.98×10^{10}	7.30×10^{8}	3.46×10^{13}	6.8×10^{11}	4.78×10^{11}	6.20×10^{9}
Reduction factor	—	82.4	92.3	11,345	—	50.7	72.5	5,587
IPC	0.551	0.549	0.555	0.554	0.274	0.276	0.260	0.253
Error	—	0.37%	0.60%	0.40%	—	0.91%	5.08%	7.57%

on the application's computation phase. In contrast, VAC shows a more balanced behavior, and we included initialization and finalization phases in the reported performance results.

Information reduction quality

To report the error that the information reduction process introduced, we analyzed the error introduced by each decision taken in this part of the methodology. Because simulation time is mainly dominated by the microarchitecture simulation time, and this simulation time is proportional to the number of instructions to simulate, we will use the number of total instructions to simulate in order to estimate simulation time. Alternatively, we also use the number of CPU bursts as an indicator of our selected representatives' quality. Finally, we use the average IPC of the computation phases to measure the error introduced by the information reduction process. Even if IPC isn't the most adequate metric to measure parallel applications' performance, it adequately represents computation phases, which in our case are free of communications.

Table 1 summarizes the results from the information reduction process. The complete trace of WRF comprises a total of 8.29×10^{12} instructions and 4.1 million CPU bursts, with an average IPC of 0.551. Simulating 10^{12} instructions is unreasonable in current microarchitecture simulators, because they normally simulate 10^{5} instructions per second, which implies nearly 10^{10} simulated instructions per day. The situation is similar with VAC, because the total number of instructions to simulate is 3.46×10^{13}. In this case, the number of CPU bursts is half a million, which indicates that the computation phases are longer than in the case of WRF. The average IPC of its computation phases is 0.274.

After applying the periodic phase detection mechanism, we identify the periodic behavior of WRF and extract two out of 160 iterations. This new trace comprises 51,000 bursts (an 82× reduction factor) and a total of 10^{11} instructions to simulate. Even if we have an 82× reduction factor in the number of instructions to simulate, the average IPC of these CPU bursts is 0.549, which represents a 0.37 percent error. In the case of VAC, we extracted two out of the 100 iterations of the detected periodic behavior. This new trace has a 50× reduction factor in CPU bursts and instructions to simulate, whereas the average IPC is 0.276, which represents a 0.91 percent error. The low error obtained in both applications is coherent with the iterative nature of HPC applications.

Afterward, the cluster analysis gathers the different CPU bursts of the trace into different clusters. This characterization process first filters a significant amount of CPU bursts (identified as noise) and then selects the clusters that cover most of the trace's execution time. In the case of WRF, we find

five clusters, covering 88 percent of the total execution cycles. These clusters represent 1,274 CPU bursts (a 3,200× reduction factor) with a total of 8.98×10^{10} instructions to simulate (a 92× reduction factor). The reduction in CPU bursts is around one order of magnitude larger than the reduction in instructions to simulate. This fact indicates that the clustering analysis effectively filters the CPU bursts that aren't representative in terms of execution time and IPC. Despite the large reduction in instructions to simulate, the average IPC of these five clusters is 0.555, which represents a 0.60 percent error. We obtained similar conclusions in the case of VAC, where only two clusters cover 81 percent of execution time, with a 700× and 70× reduction in CPU bursts and instructions to simulate, respectively, with an average IPC of 0.260, which represents a 5.08 percent error.

Finally, we must select a set of representatives from the identified clusters of CPU bursts. In the case of WRF, we select two representatives per cluster at random. Consequently, the number of selected CPU bursts is 10, and the number of instructions to simulate is 7.3×10^8 (a 11,300× reduction factor). The average IPC of these CPU bursts is 0.554, which represents only a 0.40 percent error with respect to the original trace. Because the number of identified clusters in VAC is lower, we select four representatives per cluster. Here, the total number of CPU bursts is eight, whereas the number of instructions to simulate is 6.2×10^9 (a 5,587× reduction factor). The average IPC of these CPU bursts is 0.253, which represents a 7.57 percent error with respect to the original trace.

Thus, combining these three techniques in the information reduction process leads to a reduction in simulation time between four and five orders of magnitude, with a small error in the global application's IPC. This huge reduction in simulation time with high accuracy is due to the natural behavior of HPC applications and the intelligence of the successive techniques used in this information reduction process. As a result, a parallel application that would require several years of simulation time can be simulated in a few hours, obtaining at the same time a detailed analysis at the microarchitecture and application levels.

Multilevel simulation quality

Next, we evaluate the error introduced by the two simulators involved to simulate a whole supercomputer application. We first focus on *self validation* of the methodology, or predicting the execution time of the application in the same system in which it had been executed. We then focus on *cross validation* of the methodology, or predicting the execution time of the parallel applications on a different system configuration.

Self validation. As we described earlier, we use MPsim, an in-house microarchitecture simulator with a detailed pipeline and cache hierarchy. MPsim has been developed to perform research in the processor's microarchitecture. Thus, this simulator's target architectures are future architectures that will appear in the market in about 10 years. For this reason, the accuracy when modeling a particular existing microarchitecture isn't as important as the relative differences when projecting the performance of future architectures.

To predict the performance of the chosen HPC parallel applications running in a real machine, we carefully chose the simulator parameters to model a PowerPC 970-like processor. Table 2 summarizes this machine's main characteristics.

Table 3 shows the IPC values obtained with MPsim and those measured on our real supercomputer. In the case of WRF, the IPC predictions obtained with MPsim are always within 40 percent and, on average, 25.16 percent different from the real IPC. In the case of VAC, the error per cluster remains within 40 percent, whereas the average error increases to 33.1 percent. We observe that the performance of CPU bursts over 100 million instructions is normally overestimated, whereas the performance of shorter CPU bursts is underestimated. For short traces, recovering the state of the cache hierarchy implies a significant portion of simulation time. On the real machine, part of this data is already on the cache hierarchy, which reduces the execution time of these computation phases. We've measured that

Table 2. Baseline MPsim processor configuration.

Features	Specifications
Architecture	2 cores, 2-way symmetric multithreading (SMT), superscalar architecture
Fetch, issue, and retire width	8, 5, and 5 instructions per cycle, respectively
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*
Level 1 (L1) instruction cache	64 Kbyte, direct mapped, 128-byte line, 1 cycle hit
L1 data cache	32 Kbyte, 2-way, 128-byte line, 2 cycle hit, last recently used (LRU)
L2 unified cache	1 Mbyte, 8-way, 128-byte line, 15 cycle latency, LRU
Memory latency	250 cycles
Peak memory bandwidth	2 GBps per GHz

*Two-level adaptive predictor with globally shared history buffer and pattern history. Commonly used in most current processors.

Table 3. Real IPC vs. MPsim predicted IPC comparison in self-validation experiment, using two threads per core configuration.

	WRF			VAC		
Cluster number	Real IPC	MPsim IPC	Error (%)	Real IPC	MPsim IPC	Error (%)
1	0.529	0.703	32.77	0.289	0.380	31.71
2	0.497	0.466	–6.39	0.251	0.340	35.38
3	0.618	0.429	–30.72	—	—	—
4	0.755	0.468	–38.04	—	—	—
5	0.811	0.522	–35.60	—	—	—
	Weighted average error (%)		25.16	Weighted average error (%)		33.12

it takes more than 5 million cycles. In contrast, this initialization time is less significant for longer traces. In this case, the overestimation is due to some structures optimistically modeled in MPsim: memory is assumed to be perfect (we assume that we will not access the disk). Also, some implementation details of the PowerPC 970 processor aren't public, and we've followed a best-effort approach.

Reducing the average IPC error of MPsim is outside this article's scope. More accurate simulators exist, but they're normally industrial simulators developed by the company selling the processor. These simulators are more detailed (and slow), but show IPC estimations within a 1 percent error. Because selecting the microarchitecture simulator is orthogonal to the presented simulation methodology, we report the error in execution time when using the IPC obtained with MPsim or the real IPC that would be obtained with an industrial simulator.

Finally, we simulate the original trace with Dimemas[10] using the microarchitecture simulator's feedback. Table 4 shows the reference parameters used in the different experiments. This configuration models the MareNostrum supercomputer, composed of

clusters of IBM JS21 server blades. Each node has two PowerPC 970MP processors (four cores total) and 8 Gbyte 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 these units.

Figure 2a shows the error in the total execution time of two iterations of the parallel application predicted with Dimemas, as well as of the whole application error after applying the cut factor. Using the IPC provided by MPsim, we obtain 7.67 percent error in the execution time prediction of two iterations of WRF. This error is reduced to 6.25 percent when predicting the execution time of the whole application. Using the IPC values measured on the real machine (or a highly accurate industrial simulator), the error is reduced to just 0.3 percent for two iterations and 1.62 percent for the whole application. This error is computable to the error introduced by Dimemas. In the case of VAC, the error increases to 16.2 percent (two iterations) and 21.7 percent (full application) with MPsim values, and 0.54 percent (two iterations) and 7.2 percent (full application) with the real IPC values. Thanks to the combination of communication and computation phases, the initial error of MPsim in IPC predictions is nearly divided by 3 in the final prediction of execution time.

Table 4. Baseline Dimemas cluster configuration.

Feature	Specifications
No. of nodes	2,560
Processors per node	4
Input links per node	1
Output links per node	1
No. of buses	∞*
Memory bandwidth per node	600 Mbyte/s
Memory latency	4 μs
Network bandwidth	250 Mbyte/s
Network latency	8 μs

*Contention is only defined by input and output links.

Cross validation. We can use our simulation methodology to predict the performance of a parallel application when running on a different system configuration. More specifically, we can predict the parallel application's execution time when running one task per node (single-thread configuration) instead of four tasks per node (CMP configuration). Nodes in our supercomputer infrastructure comprise two dual-core processors with a shared Level 2 (L2) cache. Furthermore, the two chips share the 8 Gbytes of memory. We've run the application using four tasks per node configuration and predicted the performance when using one task per node configuration. To cross

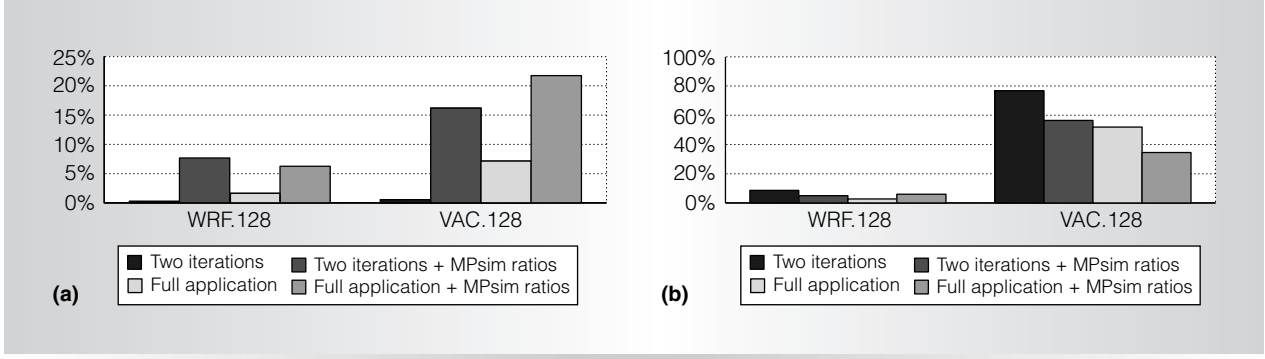

Figure 2. Execution time prediction error for Versatile Advection Code (VAC) and Weather Research Forecasting (WRF) parallel applications, for both self validation (a) and cross validation (b) experiments. The figures show the error when estimating the execution time of two iterations of the application or the full application execution time with the measured real IPC and the IPC provided by MPsim.

Table 5. WRF cross-validation of MPsim ratios, comparing the IPC running two threads per core (CMP) and a single thread per core, and the differences with the ratios in real configuration.

	Task 7			Task 9					
WRF	IPC CMP	IPC single thread	Ratio	IPC CMP	IPC single thread	Ratio	Representative average ratio	Real ratio	Error (%)
Cluster 1	0.705	0.717	1.017	0.700	0.712	1.017	1.017	1.047	–2.90
Cluster 2	0.494	0.505	1.022	0.437	0.445	1.018	1.020	1.083	–5.82
Cluster 3	0.428	0.479	1.119	0.429	0.481	1.121	1.120	1.275	–12.18
Cluster 4	0.466	0.512	1.099	0.469	0.517	1.102	1.101	1.251	–12.04
Cluster 5	0.518	0.647	1.249	0.526	0.655	1.245	1.247	1.169	6.68
							Weighted average error (%)		–4.62

Table 6. VAC cross-validation of MPsim ratios and the differences with real ratios. We express just the ratios, not the IPC, on single-thread and CMP configurations because of the higher number of representatives.

VAC	Task 51 ratio	Task 79 ratio	Task 85 ratio	Task 104 ratio	Representative average ratio	Real ratio	Error (%)
Cluster 1	1.048	1.049	1.053	1.049	1.050	1.756	–40.22
Cluster 2	1.042	1.042	1.042	1.039	1.041	1.937	–46.28
					Weighted average error (%)		–42.54

validate the results, we reran the application using one task per node and compared its runtime with the prediction.

First, we used MPsim to simulate a dual-core PowerPC 970-like machine, as described in Table 2, with just one representative of each cluster running. Then we ran two representatives of the same cluster in the same processor configuration. The performance improvement per representative is used as the required CPU ratio to the feedback Dimemas simulator. Finally, Dimemas simulations are done with the cluster parameters of the new configuration (we have just one processor per node instead of four as in Table 4).

Figure 2b shows the average error results we obtained. Without using the CPU ratios derived from MPsim, we obtain 8.6 percent and 76.7 percent errors in the execution time of two iterations of WRF and VAC, respectively. When predicting the whole application's execution time, the error is reduced to 2.78 percent and 51.9 percent for WRF and VAC, respectively. The difference in the error is due to the fact that VAC performance is much more affected than WRF when moving from a configuration with one task per node to four tasks per node. When using the CPU ratios from MPsim, we reduce the error to 4.94 percent and 56.5 percent for two iterations of WRF and VAC, respectively. In the case of WRF, the measured CPU ratios for the five representative clusters are between 1.05 and 1.21, as Table 5 shows. The predicted ratios are between 1.02 and 1.25, with an average error of 5.3 percent in the CPU ratios. As a result, the final error in the whole application's execution time prediction is just 6.04 percent.

In the case of VAC, the measured CPU ratios for the two representative clusters are 1.75 and 1.92 (see Table 6). The predicted ratios are 1.04 and 1.05, with an average error of 58.3 percent in the CPU ratios. This high error is due to the optimistic model of RAM memory implemented in MPsim. As we mentioned earlier, memory is assumed to be perfect, with no misses. VAC is suffering between nine and 11 L2 misses per kilo instruction, whereas WRF suffers only between one and two L2 misses per kilo instruction. Apart from that, VAC

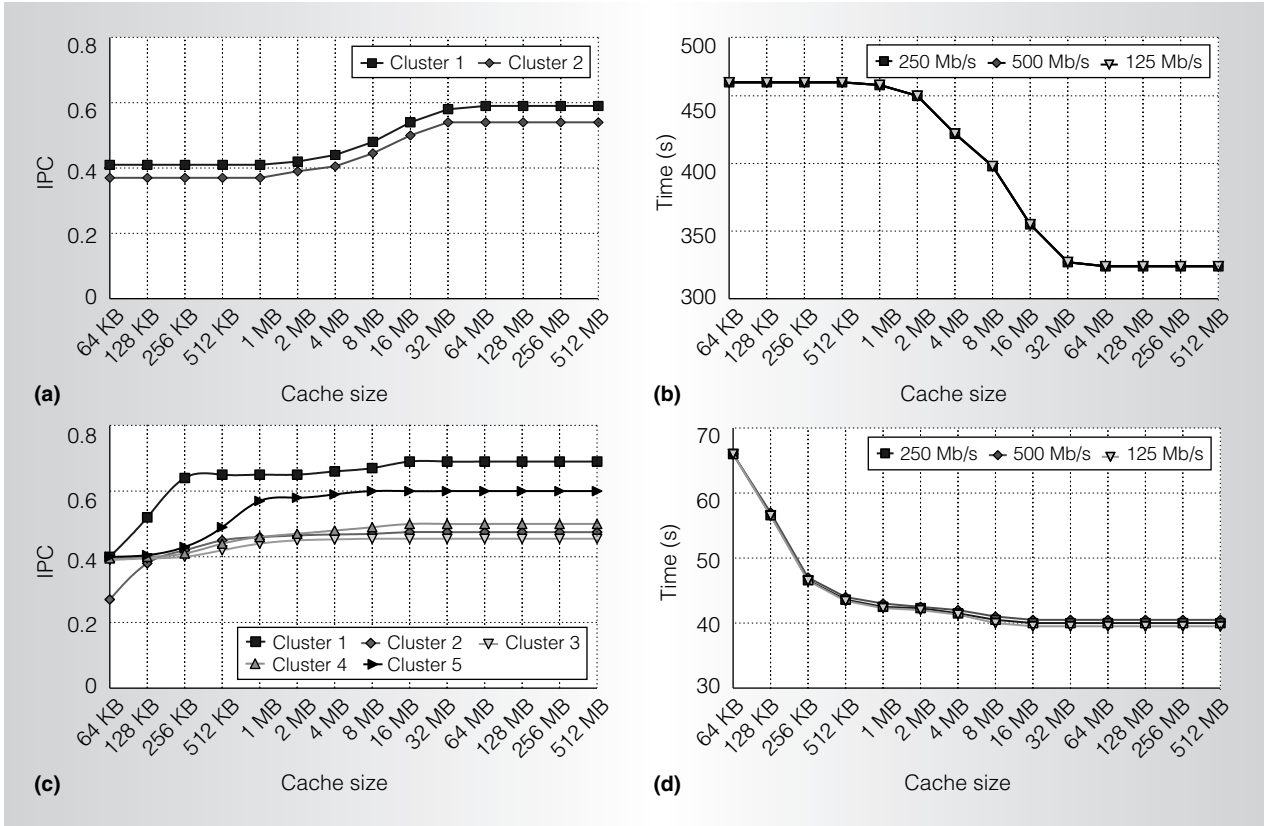

Figure 3. Performance predictions of VAC and WRF using different cache sizes and network bandwidths. We present IPC (a) and execution time (b) of VAC and the IPC (c) and execution time (d) of WRF.

suffers much more TLB misses than WRF (a 2.5× increase). As a result, the final error in execution time prediction of the whole application is 34.5 percent. Even if this accuracy is acceptable for our experiments, a more detailed model of the memory hierarchy is required in the microarchitecture simulator to obtain a more accurate prediction.

Performance analysis

This last experiment aimed to illustrate our simulation methodology's potential. In particular, we studied in detail the evolution of the behavior of the applications as the size of the L2 cache and network bandwidth increased. This kind of performance analysis is not feasible if we must simulate the whole application at the microarchitecture level, because simulating just one configuration with a given L2 size would require several years of simulation time. Instead, we obtained our results with less than 1 day of simulation.

Figure 3a shows results extracted from the VAC application in terms of IPC. In this case, we've divided the computing time into two clusters. From the perspective of the L2 cache size, we can see the same behavior in these two clusters—that is, a remarkable increase of the application's IPC if the L2 cache is larger than 2 Mbytes. The most important conclusion we can extract from this is that we can improve this application's performance by executing it on a supercomputing infrastructure whose processors have an L3 cache level that can reduce the cycles dedicated to access principal memory due to L2 misses. In Figure 3c, we depict the same results for the WRF application. In this case, we can see a strong improvement of the performance in all the clusters until 1 Mbyte of the L2 cache size is reached. This behavior is due to the small size of the input data. Thus, WRF will have enough with a small L2 cache to reach its maximum performance.

Figures 3b and 3d show the execution times obtained with Dimemas after the cluster IPC ratios. This approach lets us study the impact of architectural parameters (in this case, L2 cache size) and network parameters (in this case, bandwidth) simultaneously. According to our results, the impact of the network is negligible in the case of VAC and WRF. For these applications, the dominant performance factor is IPC, which significantly changes with the L2 cache size.

Our 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 doesn't depend on what happens with the other parallel tasks. The same isolation property can be observed on task-based programming models and DMA-based architectures.

Shared-memory programming models such as OpenMP or a mixed MPI+OpenMP application would require simulating in cycle-accurate mode all the CPU bursts executing on the shared-memory node 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.

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

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN2007-60625, by the HiPEAC European Network of Excellence, and by the IBM/BSC MareIncognito Project.

References

1. E.A. Brewer et al., ''PROTEUS: A High-Performance Parallel-Architecture Simulator,'' *Proc. ACM SIGMETRICS Joint Int'l Conf. Measurement and Modeling of Computer Systems,* ACM Press, 1992, pp. 247-248.
2. J.E. Miller et al., ''Graphite: A Distributed Parallel Simulator for Multicores,'' *Proc. 16th IEEE Int'l Symp. High Performance Computer Architecture,* IEEE Press, 2010, doi:10.1109/HPCA.2010.5416635.
3. E. Argollo et al., ''COTSon: Infrastructure for Full System Simulation,'' *ACM SIGOPS Operating Systems Review,* vol. 43, no. 1, 2009, pp. 52-61.
4. E.S. Chung et al., ''A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs,'' *Proc. 16th Int'l ACM/SIGDA Symp. Field Programmable Gate Arrays,* ACM Press, 2008, pp. 77-86.
5. E. Perelman et al., ''Using SimPoint for Accurate and Efficient Simulation,'' *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* ACM Press, 2003, pp. 318-319.
6. R.E. Wunderlich et al., ''SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,'' *ACM SIGARCH Computer Architecture News,* vol. 31, no. 2, 2003, pp. 84-97.
7. M. Casas, R.M. Badia, and J. Labarta, ''Automatic Structure Extraction from MPI Applications Tracefiles,'' *Proc. Int'l Euro-Par Conf. Parallel Computing,* LNCS 4641, Springer, 2007, pp. 3-12.
8. M. Ester et al., ''A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,'' *Proc. 2nd Int'l Conf. Knowledge Discovery and Data Mining,* AAAI Press, 1996, pp. 226-231.
9. J. Gonzalez, J. Gimenez, and J. Labarta, ''Automatic Detection of Parallel Applications Computation Phases,'' *Proc. IEEE Int'l Symp. Parallel & Distributed Processing,* IEEE CS Press, 2009, doi:10.1109/IPDPS.2009.5161027.
10. S. Girona, J. Labarta, and R.M. Badia, ''Validation of Dimemas Communication Model for MPI Collective Operations,'' *Proc. 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* Springer, 2000, pp. 39-46.
11. G. Toht, ''Versatile Advection Code,'' *Proc. Int'l Conf. Exhibition on High-Performance Computing and Networking,* Springer, 1997, pp. 253-262.
12. J. Michalakes et al., ''The Weather Research and Forecast Model: Software

Architecture and Performance,'' *Proc. 11th ECMWF Workshop Use of High Performance Computing in Meteorology,* World Scientific Books, 2004, pp. 156-168.

Juan Gonzalez is a researcher at the Barcelona Supercomputing Center and a PhD student in the Computer Architecture Department of the Polytechnic University of Catalonia. His research interests include the application of data-mining techniques to the automatic performance analysis of parallel applications. Gonzalez has an MS in computer science from the Polytechnic University of Catalonia.

Marc Casas is a post-doctoral research scholar at the Lawrence Livermore National Laboratory. His research interests span parallel computing, focusing on fault-tolerance and reliability of HPC applications, scalability, and automatic performance analysis. Casas has a PhD in computer architecture from the Polytechnic University of Catalonia.

Judit Gimenez is a research manager at the Polytechnic University of Catalonia and the Performance Tools Group manager at the Barcelona Supercomputing Center. Her research interests include parallel computing, focusing on developing performance analysis tools. Gimenez has an MS in computer science from the Polytechnic University of Catalonia.

Miquel Moreto is a lecturer in the Computer Architecture Department at the Polytechnic University of Catalonia. His research interests include modeling parallel computers and resource sharing in multithreaded architectures. Moreto has a PhD in computer architecture from the Polytechnic University of Catalonia.

Alex Ramirez is an associate professor at the Polytechnic University of Catalonia and a research manager at the Barcelona Supercomputing Center. His research interests include heterogeneous multicore architectures, hardware support for programming models, and simulation techniques. Ramirez has a PhD in computer science from the Polytechnic University of Catalonia.

Jesus Labarta is a professor at the Polytechnic University of Catalonia and director of the Computer Science Department of the Barcelona Supercomputer Center. His research interests include high-performance architectures and system software, particularly programming models and performance tools. Labarta has a PhD in telecommunications from the Polytechnic University of Catalonia.

Mateo Valero is a professor at the Polytechnic University of Catalonia and director of the Barcelona Supercomputer Center. His research interests include high-performance architectures. Valero has a PhD in telecommunications from the Polytechnic University of Catalonia. He's a fellow of IEEE and the ACM and a Distinguished Research Fellow at Intel.

Direct questions and comments about this article to Juan Gonzalez, Campus Nord UPC — C6 Building Office 002, c/Jordi Girona, 1-3, 08034 Barcelona, Spain; juan.gonzalez@bsc.es.