Servet: A Benchmark Suite for Autotuning on Multicore Clusters

Jorge González-Domínguez, Guillermo L. Taboada, Basilio B. Fraguela, María J. Martín, Juan Touriño
Computer Architecture Group
Department of Electronics and Systems
University of A Coruña, Spain
Email: {*jgonzalezd, taboada, basilio.fraguela, mariam, juan*}*@udc.es*

***Abstract*—The growing complexity in computer system hierarchies due to the increase in the number of cores per processor, levels of cache (some of them shared) and the number of processors per node, as well as the high-speed interconnects, demands the use of new optimization techniques and libraries that take advantage of their features.**

In this paper Servet, a suite of benchmarks focused on detecting a set of parameters with high influence in the overall performance of multicore systems, is presented. These benchmarks are able to detect the cache hierarchy, including their size and which caches are shared by each core, bandwidths and bottlenecks in memory accesses, as well as communication latencies among cores. These parameters can be used by autotuned codes to increase their performance in multicore clusters. Experimental results using different representative systems show that Servet provides very accurate estimates of the parameters of the machine architecture.

***Keywords*-Benchmarking, Autotuning, Multicore Clusters, Optimization.**

I. Introduction

Nowadays, there is a trend towards developing autotuned codes, which can automatically optimize their performance depending on the machine on which they are executed. Some tools have already been developed for sequential computation [1]–[3] using a wide search mechanism to find the most appropriate algorithm, although the knowledge of some hardware characteristics can reduce the search time [4].

Additionally, many optimization techniques have been developed for parallel computing. Among the different parallel architectures, clusters of multicores pose significant challenges, as they present a hybrid distributed and shared memory architecture with several hierarchies determined by non-uniform communication latencies. Furthermore, because of sharing memory or even cache among some cores, concurrent memory accesses using different threads can decrease the memory access performance per core.

Two main approaches are being studied to improve the performance of parallel applications in clusters of multicores. One consists in implementing codes that take into account the system characteristics [5]–[8], for instance minimizing the number of messages across the network or using blocks of data that fit in the caches to avoid cache misses. The other approach maps the processes to specific cores to improve the performance without changing the codes [9]–[11]. Knowledge of the topology of the machine and of some hardware parameters is necessary in both approaches.

System parameters and specifications are usually vendor-dependent and often inaccessible to user applications. For instance, the administration tool *dmidecode* reports information about the hardware as described in the BIOS, but it is restricted to system administrators. Therefore, estimation by benchmarks is the only general and portable way to find out the hardware characteristics, without worrying about the vendor, the OS or the user privileges. Besides, this approach provides experimental results about the performance of the systems, obtaining a more reliable estimate than inferring them from the machine specifications.

This paper presents Servet[1], a portable benchmarking suite to obtain the relevant hardware parameters of clusters of multicores and, hence, support the automatic optimization of parallel codes on these architectures. Cache size and hierarchy, bottlenecks in memory access and communication overheads are included among the estimated parameters. The accuracy of the estimates has been satisfactorily validated on several machines with very different architectures.

The rest of the paper is organized as follows. Section II presents previous works focused on obtaining system parameters to support autotuning. Section III describes the benchmarks. Section IV summarizes the experimental environments and the results. Finally, concluding remarks are presented in Section V.

II. Related Work

The automatic extraction of system parameters to support the autotuning of parallel applications is a topic which has been previously tackled with different approaches. The X-Ray tool [12] provides micro-benchmarks to automatically obtain some characteristics of the CPU and the cache hierarchy for unicore systems. In [13] Yotov et al. presented a new methodology to obtain the cache parameters. However, they reported some issues measuring the characteristics of the cache levels lower than L1 because they are physically indexed. In

[1]As this suite dissects the machines to discover their characteristics, it obtains its name from Miguel Servet, a Spanish theologian, physician, cartographer and humanist who lived in the XVIth Century and performed many dissections, being the first European to describe the function of pulmonary circulation.

their benchmarks contiguous memory allocation is compulsory, therefore, they must request virtual memory backed by a superpage. As the way to request it depends on the OS, their approach is not portable. Servet provides an algorithm to detect the sizes of all cache levels (and whether they are shared) in any system with any OS (either using page coloring or not), thus solving the main problem detected in previous works [12], [13]. As we will see, our approach also addresses the influence of hardware prefetchers in the measurements.

P-Ray [14] extended X-Ray benchmarks to detect the characteristics of multicore systems. Although it is very appropriate for SMPs, it inherits most of the issues of X-Ray (e.g., non-portable L2/L3 cache parameters determination) and lack some relevant parameters for multicore clusters like communication overhead. Although the algorithm to detect the processor mapping in P-Ray could be used (with manual work) to show different communication latencies on multicore systems, it is restricted to shared memory systems, so it is not applicable to clusters. Our benchmark automatically detects all communication layers in multicore clusters. Besides, Servet provides additional information which has not been addressed in previous works, such as the scalability of each layer and its behavior according to the message size. This information can be very useful to optimize applications in multicore clusters, where the behavior of the networks can be a very important bottleneck. Another shortcoming of P-Ray is that it assumes a uniform cost in the intra-node memory access. Our experimental results show in Section IV that in practice it is not true: cores that share the memory bus present a greater overhead than the other cores in the same cell, and cores in the same cell present a greater overhead than the other cores in the same node. Servet is able to detect these types of issues, which are very important for mapping policies, by using the benchmark to automatically detect which cores present an overhead when accessing concurrently to memory and the magnitude of this overhead.

Some tools to optimize parallel codes in multicore clusters according to their architecture have been developed [9], [10]. They are focused on communication overheads, trying to obtain the best process placement policies by increasing the use of shared memory, thus avoiding messages across the cluster interconnection network. However, they do not take into account the effective memory bandwidth, specially when concurrent accesses from different threads are present. Furthermore, although these tools rely on the characteristics of the machine, they do not actually measure them. *MPIPP* [9] needs the communication bandwidth among cores and obtains it from the technical specification of the machine, but this is not usually possible. Mercier and Clet-Ortega [10] obtain the topology of the machine from its specifications and then estimate the communication costs according to them. Although the knowledge about the topology is often available from machine documentation, this is a non-portable approach. Additionally, these estimates are usually poorly accurate.

Summarizing, our approach improves previous works by providing a portable estimate of several parameters of the machine architecture such as the cache size (L1, L2 and, when available, L3 caches), cache hierarchy (determining if a cache is private to a particular core or shared among several cores), shared memory access bandwidth, shared memory hierarchy (cores that share memory and NUMA system hierarchy) and communication overheads.

III. Benchmarks

The benchmarks implemented in Servet to determine cache size, cache sharing topology, memory performance and communication overheads are next described.

A. Cache Size Estimate

The knowledge of the cache size is used in many optimization techniques to divide the computation in blocks of data which fit in cache in order to minimize the number of cache misses and, therefore, increase the memory access performance.

Several benchmarks that estimate the cache size have been proposed during the last years. We have followed the approach of Saavedra and Smith [15]: measuring the number of cycles used to traverse arrays of different sizes using 1KB strides. This stride has been chosen because it is big enough to avoid influences of the hardware prefetcher on the measured number of cycles, as current prefetchers work with strides up to 256 or 512 bytes. It is also larger than any existing cache line size and it is a divisor of any cache size.

However, this approach presents several drawbacks [13]: the results may be disturbed by unintended optimizations of aggressive compilers and they must be interpreted to determine the different cache sizes. Our code improves this benchmark using values read from an array as stride, thus avoiding aggressive compiler optimizations, and providing directly the cache sizes.

The algorithm used (*mcalibrator*) is shown in Figure 1. Its outputs are two arrays `S` and `C`, of length `n`, containing the sizes of the traversed arrays and the average number of cycles required by each access during their traversal, respectively.

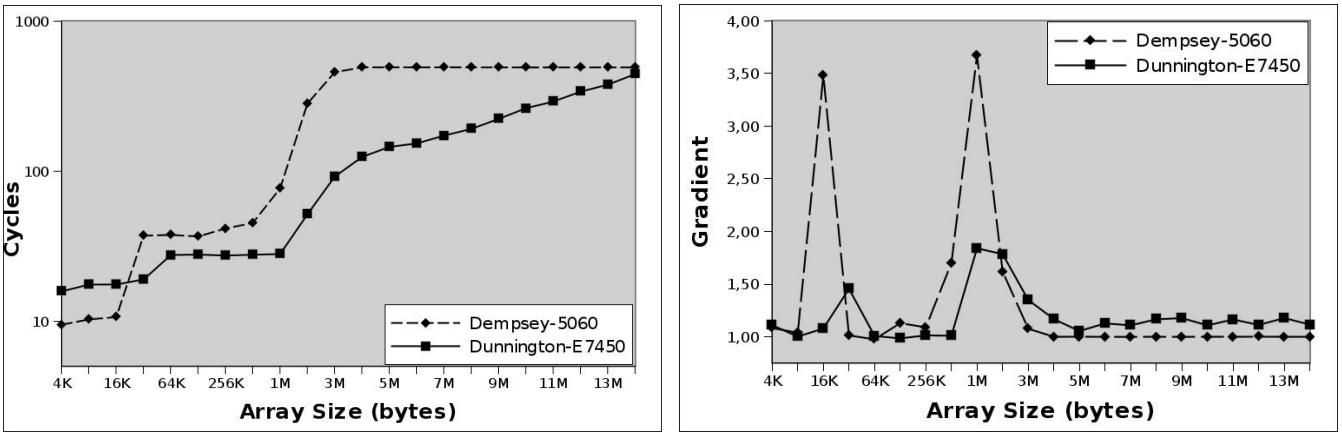
Figure 2(a) presents the cycles obtained with this algorithm on two Intel Xeon based architectures (*Dempsey*-5060 and *Dunnington*-E7450), whereas Figure 2(b) shows the gradient of the previous results, that is $C[k+1]/C[k], 0 \leq k < n$. These architectures will be used to explain the algorithm to detect the cache hierarchy from the cycles of access to memory and their corresponding gradients.

1) L1 Cache: Traversing an array with all its elements in cache is faster than traversing one which does not fit. Therefore, the sizes where the number of cycles rises indicate that they do not fit in cache and cache misses appear. Sharp changes in the slope of the cycles graph correspond to peaks in the gradients one. The L1 cache size is determined by the first peak of the gradients: 16KB for *Dempsey* and 32KB for *Dunnington* according to Figure 2.

2) Lower Cache Levels: The approach followed to estimate the L1 cache size cannot be always applied to lower levels. L1 caches are typically virtually indexed, but lower levels are always physically indexed for a number of practical reasons [16]. This is a problem when the cache size is larger than one page because contiguity in virtual memory does not imply

```
aux = 0  // Auxiliary variable to help to avoid compiler influences
i = MIN_CACHE
n = 0 // Number of cache sizes tested
while i≤MAX_CACHE do
    S[n] = i  // Size of the traversed array
    size = The amount of integers stored in S[n] bytes
    for j=0;j<size;j=j+1 do
        A[j] = The amount of integers stored in 1KB  // Each position keeps the stride
    end
    // The access to the array is in the loop to know the stride
    for j=0;j<size;j=j+A[j] do
        aux = aux + size  // A variable update to avoid compiler optimizations
    end
    C[n] = The number of cycles to perform the previous loop
    n=n+1
    if i<2MB then
        i=i*2
    else
        i=i+1MB
    end
end
```

Fig. 1: *mcalibrator* algorithm



(a) Cycles needed to traverse an array

(b) Gradient of the rise of cycles

Fig. 2: *mcalibrator* results (run on two cores)

adjacency in physical memory, which leads to the generation of misses in tests with arrays much smaller than the cache considered. Therefore, peaks in the gradient function could not be enough to estimate the cache size, as for *Dempsey*, with high gradient values in the range [512KB,2MB]. Although some OSs solve this problem applying page coloring, others such as Linux, widely used in scientific computing, do not. As commented in Section II, an alternative has been proposed by Yotov et al. [13] but their solution is not portable.

Our benchmark overcomes this problem following a probabilistic approach. Since the OS can map a virtual page to any physical page, no assumptions can be made on which cache sets of a physically indexed cache correspond to a given virtual page. Now, in a K-way physically indexed cache of size CS with a page size PS every cache way can be divided in $CS/(K*PS)$ page sets, that is, groups of cache sets that can receive data from the same page. As a result, if the probability a given virtual page is mapped to a given page set is uniform, the number of pages X per page set belongs to a binomial $B(NP, (K*PS)/CS)$, where NP is the number of pages involved in the access [8]. Since each set can contain up to K pages without conflicts, the probability $P(X > K)$ is the expected miss rate during the repeated access to the NP pages.

Thus, based on the outputs of *mcalibrator* (Figure 1), the algorithm in Figure 3 can be used to determine L2 and L3 (if present) cache size according to the previous reasoning. The probabilistic algorithm starts calculating the number of pages and the miss rate for each *mcalibrator* result. After that, it calculates the divergences between the measured and the predicted miss rates according to the binomial distribution.

$hit\_time = MIN(C)$;
$miss\_overhead = MAX(C) - MIN(C)$
for *i=0;i<n;i=i+1* **do**
  $MR[i] = (C[i] - hit\_time)/miss\_overhead$ `// Miss Rate`
  $NP[i] = S[i]/PS$ `// Number of Pages`
end
foreach *tentative cache size CS and associativity K* **do**
  $div[CS][K] = 0$
  for *i=0;i<n;i=i+1* **do**
    $div[CS][K]$ = $div[CS][K]$ + | $MR[i]$ - $P(X > K)$ |, $X \in B(NP[i], (K * PS)/CS)$
  end
end
Result: The statistical mode of *CS* using the five elements of *div* with the lowest values

Fig. 3: Probabilistic algorithm to determine the size of physically indexed caches (L2, L3) based on *mcalibrator* outputs

Data: *mcalibrator* outputs from MIN_CACHE to MAX_CACHE
foreach *peak in the gradients* **do**
  if *This is the first peak* **then**
    Estimate the L1 cache size using the peak position
  else
    if *Peak is related only to a single array size* **then**
      Estimate the corresponding cache size using the peak value
    else
      Estimate the corresponding cache size from the probabilistic algorithm using *mcalibrator* outputs where gradient is larger than 1 around the peak
    end
  end
end
if *The largest array sizes show a gradient* > 1 **then**
  The corresponding cache size is the estimate of the probabilistic algorithm using *mcalibrator* outputs with the largest sizes
end

Fig. 4: Algorithm to detect the cache levels and their sizes

The estimated cache size is the one with the highest similarity (less divergence).

The accuracy of this algorithm is higher than only searching peaks in the gradients of the *mcalibrator* outputs. For instance, in the *Dempsey* case, a 1MB L2 cache would be erroneously estimated looking at the *mcalibrator* outputs. The estimate of the latter algorithm analyzing the [256KB,4MB] range is 2MB, the correct value.

Besides, this algorithm is able to provide a correct cache size estimate when gradients are higher than 1 for a wide size range. This is the case of *Dunnington* (see Figure 2): the use of the algorithm in the range [3MB,14MB] provides 12MB as result, which is the actual L3 cache size.

The overall algorithm to detect the number of levels of cache and their sizes is presented in Figure 4. As L1 caches are virtually indexed, their size is always calculated using the first peak of the gradients. However, there are two ways to estimate the size of the next levels. A peak clearly located only in one array size means that the OS has used page coloring so the behavior of the cache is analogous to that of a virtually indexed one and the position of the peak determines the cache size. However, a peak with high gradients for several array sizes needs the use of the probabilistic algorithm. Therefore, this benchmark is completely portable, being independent from the application of page coloring by the OS. Finally, the probabilistic algorithm is used again if gradients are higher than 1 for the largest arrays.

B. Determination of Shared Caches

The knowledge of which cores share a concrete cache level can be useful in order to speed up the memory accesses. On the one hand, if two processes work with the same block of data which fits in cache, mapping them to cores which share cache would improve the performance, as they could exchange data using the cache. On the other hand, if they do not work with the same data, their working sets could not fit in a shared cache, leading to more replacements and misses. In this case scheduling techniques for autotuning would map the processes to cores that do not share cache in order to minimize the misses.

```
Data: l, number of cache levels, and CS[0..l-1] cache size per level
for i=0;i<l;i=i+1 do
    P_sc[i] = Empty list
    ref = Cycles obtained from mcalibrator run on one core using an array of size (2/3) * CS[i]
    foreach pair of cores in the system do
        c = Cycles obtained from mcalibrator run in parallel on the cores of the pair using an array of size
            (2/3) * CS[i] in each core
        ratio = c/ref
        if ratio>2 then
            Add the pair to P_sc[i]
        end
    end
end
Result: P_sc[0..l-1]
```

Fig. 5: Algorithm to determine the shared caches

Figure 5 shows our benchmark to detect shared caches. The inputs are the number of cache levels l and an array CS, of length l, with the cache size per level. For each cache level i, the first step is to call *mcalibrator* using an array of size a little larger than $CS[i]/2$ and keep the result as reference. Then *mcalibrator* is invoked simultaneously on two arrays of this size in two threads, varying the cores where the threads are mapped. The chosen array size provokes that two arrays do not fit simultaneously in cache so, when cores share cache, the array created by each of them is replacing the other one and the number of cycles increases. The output is an array of lists P_{sc} (one list per cache level) with the pairs with a number of cycles at least twice greater than the reference value (metric ratio > 2 in Figure 5), and therefore whose cores share cache.

C. Memory Access Overhead Characterization

When several cores share the main memory, performance bottlenecks may arise with concurrent memory accesses. The knowledge of these bottlenecks would allow to implement scheduling policies in autotuned applications to avoid them and improve the memory access performance.

A benchmark that provides performance results of concurrent memory accesses has been developed. This benchmark is similar to the previous one, as it compares the bandwidth to memory using an isolated core (reference) with the one obtained when accessing by pairs. Our approach to calculate the bandwidth is based on similar tools which measure it, such as *STREAM* [17]. In our case, it is the bandwidth from the copy of all the elements stored in one array to another (these arrays must not fit in cache).

The algorithm of the benchmark is shown in Figure 6. For each pair of cores, the bandwidth of one core when both of them are concurrently accessing to memory is calculated and compared to the reference value, it means, the memory bandwidth when accessing with an isolated core. A bandwidth for concurrent accesses significantly lower than the reference indicates an overhead.

However, distinguishing the different magnitudes of overhead and which pairs suffer each of them is also interesting. To do it, the algorithm works with two arrays: BW, with the different bandwidths lower than the reference value, and P_m, which contains the pairs of cores which cause each overhead ($P_m[i]$ is the list of the core pairs which obtained a concurrent bandwidth similar to $BW[i]$). When a bandwidth lower than the reference value is found, the algorithm searchs in the BW array if any previous pair already obtained a similar overhead. In this case, the pair which is being studied is added to the list of P_m corresponding to that bandwidth. Otherwise, this is the first pair with that specific overhead, so BW and P_m are updated appending to them a new entry with the new bandwidth and a list with the studied pair, respectively.

```
n = 0  // Number of different overhead levels found
ref = Memory bandwidth when accessing with an isolated core
foreach pair of cores in the system do
    b = Memory bandwidth for one core when accessing both cores concurrently
    if b<ref then
        if b is similar to a given BW[i], 0≤i<n then
            Add the pair to P_m[i]
        else
            BW[n] = b
            P_m[n] = Empty list
            Add the pair to P_m[n]
            n=n+1
        end
    end
end
Result: n, BW[0..n-1], P_m[0..n-1]
```

Fig. 6: Algorithm to characterize the memory access overhead

The groups of cores that collide accessing to memory with a given overhead are easily obtained from P_m. For instance, if the list in $P_m[i]$ has the pairs (0,1),(0,2),(3,4) and (3,5), it allows to identify two groups for the overhead $BW[i]$: {0,1,2}

and {3,4,5}.

This knowledge about memory access overhead of groups of cores can be used to analyze the scalability of the memory access performance. This parameter has an special importance, as autotuning could optimize codes by limiting the number of cores accessing to memory if a poorly scalable memory system is detected. Characterizing the effective bandwidth according to the number of threads that are being executed only requires one group per overhead. For instance, for the previous example, the concurrent memory access bandwidth of the cores 0, 1 and 2 is the same as the one for cores 3, 4 and 5. Therefore, using the arrays BW and P_m, the effective bandwidth to memory can be characterized without using all cores.

D. Determination of Communication Costs

Currently, many optimization techniques that can be used in autotuned applications for clusters of multicores are focused on reducing the communication overhead taking into account the different latencies [9], [10].

Traditionally, the characterization of the communication overhead has been done using extensions either of the LogP model [18] or of the Hockney's linear model [19]. However, both of them show poor accuracy on current communication middleware on multicore clusters [20], which implement several communication protocols, both depending on the message size and the communication layer (e.g., high-speed networks, inter-process and intra-process shared memory transfers).

In this section a new benchmark which provides a more accurate communication overhead characterization is proposed. The characterization provided by our benchmark is divided in three parts. Firstly, communication layers (sets of pairs of cores whose communication costs are similar) are established. Then, these layers are used to characterize communication performance and, finally, to evaluate the scalability of the communication system.

In order to group the cores according to their communication costs, the benchmark shown in Figure 7 has been implemented. The reference implementation uses MPI. It compares the latencies when sending a message between different pairs of cores. Several representative message sizes can be selected for this task. In our case, the size of the message is equal to the L1 cache size, because it allows to find differences in communications when sharing other cache levels.

The algorithm is similar to the previous one: for each pair of cores, it obtains the latency to send a message between them and stores the different latencies in the array L. Besides, the array P_l is created so that $P_l[i]$ keeps the list with the pairs with a latency $L[i]$. Finally, the cores that present the same communication performance are grouped.

Once the layers are established, the followed approach is to store the performance results of a micro-benchmarking of a point-to-point communication (also implemented with MPI) for representative message sizes and for each representative pair of cores (one per layer). The communication performance for the rest of the pairs is the same as for the representative pair of their group.

```
n = 0  // Number of different layers
foreach pair of cores in the system do
    l = Latency sending a message between the two cores
    if l is similar to a given L[i], 0≤i<l then
        Add the pair to P_l[i]
    else
        L[n] = l
        P_l[n] = Empty list
        Add the pair to P_l[n]
        n=n+1
    end
end
Result: n, L[0..n-1], P_l[0..n-1]
```

Fig. 7: Algorithm to categorize the communication costs

Finally, in order to characterize the scalability of an interconnect, the performance of all the cores in a given layer concurrently sending one message is compared to the latency of an isolated message. In fully scalable communication systems, times should be similar because each core only sends one message. However, many cluster interconnection networks can present performance penalties when concurrent messages are sent through them. Sending concurrently `N` messages of size `S` usually costs more than sending one message of size `N*S`. Thus, it is possible to optimize the communication performance by gathering messages in poorly scalable systems.

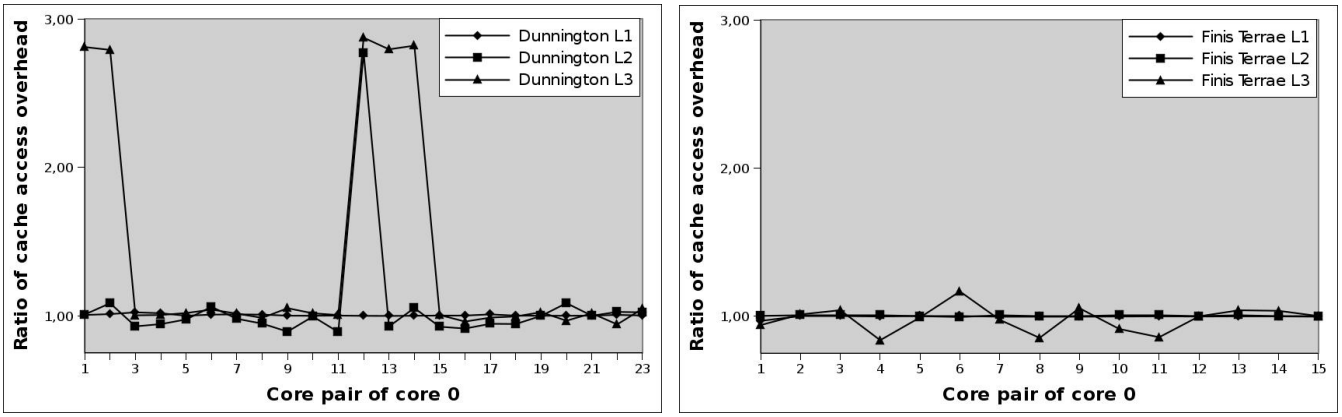
IV. Experimental Evaluation

Two systems have been used to test the benchmarks presented in this paper. The first one is a machine with four *Dunnington* Intel Xeon E7450 hexacore processors (2,40 GHz). The six cores in the same processor share a 12MB L3 cache. There are also 3MB L2 caches shared by pairs of cores and individual 32KB L1 caches. The MPI implementation is MPICH2 version 1.1.1.

The second machine is the Finis Terrae supercomputer [21]. It consists of 142 HP RX7640 nodes, each of them with 8 Itanium2 Montvale dual-core processors (1,60 GHz), hence 16 cores, distributed in two cells (each cell has 4 processors and 8 cores). Each cell has its own 64GB main memory, so all cores in the same node logically share 128GB of memory. Memory accesses are performed across buses shared by pairs of processors. Each core has its individual 16KB L1, 256KB L2 and 9MB L3 caches. Nodes are interconnected via InfiniBand (20 Gbps). The MPI library is HP MPI 2.2.5.1, with a shared memory device (SHM) and an InfiniBand device (IBV).

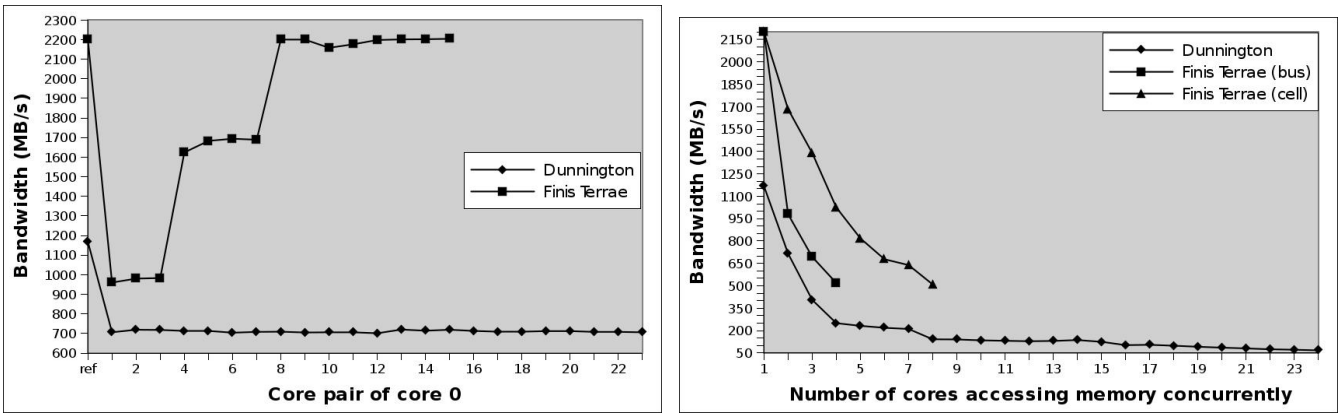
A. Cache Size Estimate

As the cache size estimates do not require a multicore cluster to be tested, two additional systems were used to prove the correctness of the approach shown in Section III-A: *Dempsey*, an Intel Xeon 5060 dualcore processor (3,20 GHz) with 16KB L1 and 2MB L2 caches, and the unicore AMD Athlon 3200 (2 GHz) with 64KB L1 and 512KB L2 caches.



(a) *Dunnington* (b) *Finis Terrae*

Fig. 8: Results from the shared cache benchmark



(a) Memory access performance with two simultaneous accesses (b) Memory access performance with multiple simultaneous accesses

Fig. 9: Results from the memory overhead benchmark

These systems present a good variety of characteristics: Intel and AMD processors, 3 or only 2 cache levels, shared and individual caches, etc.

The benchmark presented in Section III-A was tested in these four machines (10 cache sizes in total) and all the estimates agreed with the specifications.

B. Determination of Shared Caches

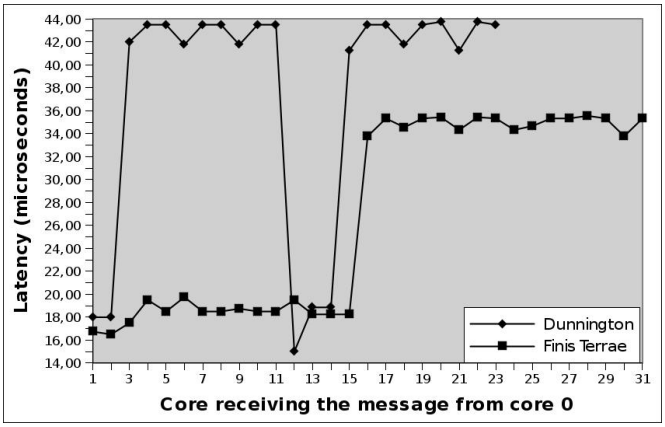
Figure 8(a) illustrates the results for the detection of which caches are shared (Section III-B) in the *Dunnington* system, whereas Figure 8(b) shows the *Finis Terrae* numbers, obtained using one node. For clarity purposes, only the results obtained with the pairs that contain core 0 are shown. The performance metric in the graphs is the ratio of cache access overhead presented in the benchmark (see *ratio* in Figure 5).

In *Dunnington*, core 0 shares its L2 cache with core 12 and the L3 cache is shared by all the cores in the same hexacore processor, cores {0,1,2,12,13,14}. In principle, one would expect that the OS would number the cores according to their physical layout. Thus, it would be expected that cores 0 and 1 share the L2 cache and that the 6 cores sharing the L3 cache would be numbered 0-5. This benchmark is useful to discard this kind of assumptions and show the actual topology. In *Finis Terrae* all the ratios are below 2, so the benchmark detects that all the caches are private.

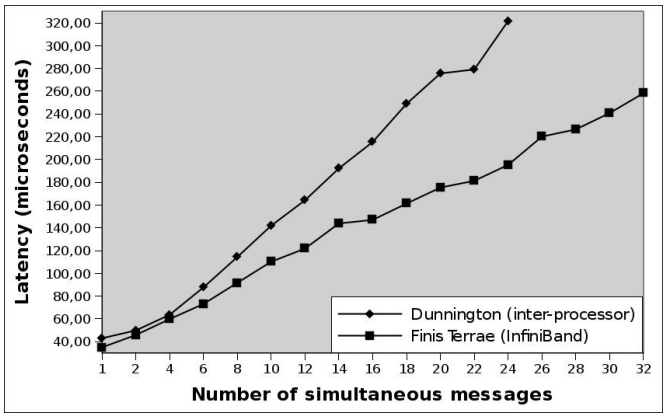
C. Memory Access Overhead Characterization

Figure 9(a) illustrates the memory bandwidth when two cores are accessing memory concurrently in the *Dunnington* and *Finis Terrae* systems. For clarity purposes only pairs of cores that contain core 0 are shown. Besides, the bandwidth achieved individually by the core 0 (the reference value) is labeled as *ref* in the *x* axis.

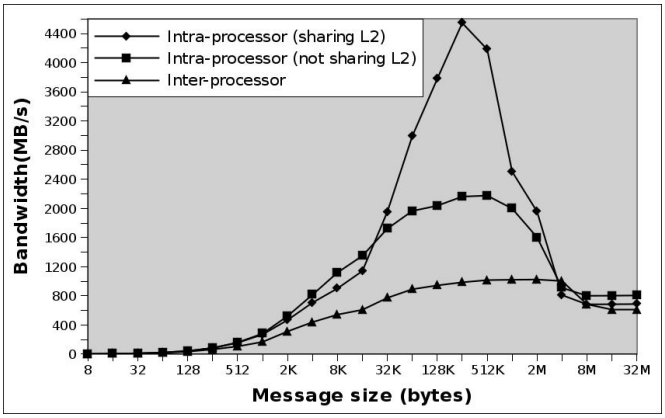
Dunnington presents overhead when two cores access memory simultaneously, and its magnitude is the same independently of the pair of cores. However, the results in *Finis Terrae* depend on the cores tested. Core 0 accessing memory concurrently with cores 1, 2 or 3 shows the lowest bandwidth. The reason for this behavior is that, according to the system specification, these cores belong to processors which share the bus to memory. A higher bandwidth, but still a 25% lower than the reference value, is achieved from cores 4 to 7, which is explained by the fact that these cores are located in the same cell as core 0 and present some performance degradation from sharing the cell memory. Finally, there is



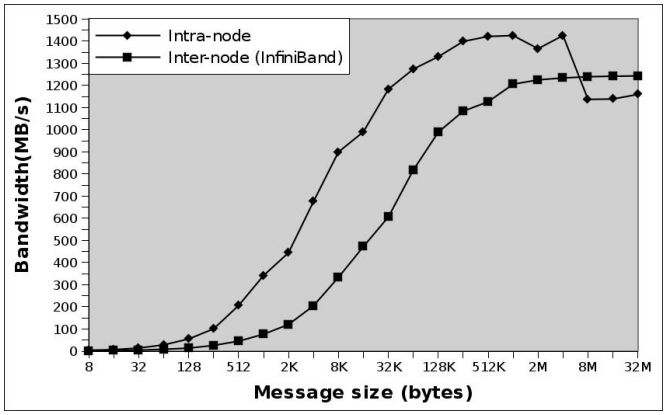
(a) Message-passing latency (L1 message size)



(b) Latency scalability (L1 message size)



(c) Point-to-point bandwidth (*Dunnington*)



(d) Point-to-point bandwidth (*Finis Terrae*)

Fig. 10: Results from the communication costs benchmark

no particular overhead when cores from different cells are accessing memory simultaneously.

Figure 9(b) shows the effective bandwidth when several cores access memory concurrently. Only groups whose cores presented overhead in the left graph must be analyzed, thus for the *Finis Terrae* case there are two lines: *bus*, for cores that share the bus to memory, and *cell*, for cores located in the same cell.

D. Determination of Communication Costs

Figure 10(a) illustrates the latency of MPI message transfers between two cores (the affinity of MPI processes to particular cores has been set with the *sched* system library). In this case *Finis Terrae* results are shown for two nodes (hence, up to 32 cores), which is enough to characterize all the different communication costs. For the purpose of clarity, only communications whose source core is 0 are shown.

In *Dunnington*, communications among cores in the same hexacore processor present lower latencies than inter-processor transfers, especially when sharing the L2 cache (core 12). In *Finis Terrae* the intra-node communications (cores 1 to 15) are around two times faster than inter-node ones (cores 16 to 31). This benchmark not only provides the hierarchy of the system according to the communication overhead, but also the actual communication performance.

Figure 10(b) shows the scalability of the inter-processor transfers for the *Dunnington* system and the inter-node transfers (InfiniBand) for the *Finis Terrae* through the latency of several messages sent concurrently across these interconnects. The results show a moderate scalability. For instance, a message sent through the InfiniBand network in *Finis Terrae* when there are other 31 messages is 7 times slower than a message sent alone.

Figures 10(c) and 10(d) present the point-to-point bandwidth achieved by representative pairs of cores, one per each layer detected using the benchmark shown in Figure 7. These results allow to accurately estimate the communication overhead of a given message, both taking into account the communication layer being used (e.g., InfiniBand, intra-processor, inter-processor), and the message size. Thus, using this benchmark an autotuned code can analyze the cost of a communication (and its alternatives) beforehand.

E. Execution Times

Regarding the execution times of the benchmarks, they are very dependent on the target architecture, specially on the total numbers of cores, the number of caches and the complexity of the communication layers. As a reference, Table I shows the execution times of each benchmark in the two multicore clusters. Anyway, as these benchmarks do not use dynamic

information, they must be run only once at installation time, and each time the system changes its hardware configuration (e.g., if new nodes are added to the system). The information obtained can be stored in a file to be consulted by the applications to guide optimizations when needed, thus their execution time is not important.

TABLE I: Execution times (in minutes) of all the benchmarks

	Dunnington	*Finis Terrae*
Cache Size Estimate	2'	2'
Determination of Shared Caches	11'	3'
Memory Access Overhead	20'	5'
Communication Costs	22'	33'
Total	55'	43'

V. Conclusions

This paper has presented Servet, a fully portable suite of benchmarks to obtain the most interesting hardware parameters to support the automatic optimization of applications on multicore clusters. These benchmarks determine the number of levels of cache, their sizes, the cache topology of shared caches, the memory access bottlenecks, and the communications scalability and overheads. Our tests prove that the suite provides highly accurate estimates according to the system architecture specifications.

Many optimization techniques can take advantage of the hardware parameters determined. The information about the possible overheads can be used to automatically map the processes to certain cores in order to avoid either communication or memory access bottlenecks. Even if not all overheads can be avoided, mapping policies with a good trade off according to the characteristics of the code can be applied. In some cases it could be even better not to use some cores to avoid performance drops.

Not only the mapping techniques can take advantage of Servet. Tiling is one of the most widely used optimization techniques and our suite can help to this technique by providing all the cache sizes in a portable way. Furthermore, many programs provide several implementations of parts of their code in order to take advantage of different architectures. Using the system parameters obtained by Servet it is possible to adapt the behavior of an application to maximize its performance.

Servet is publicly available under GPL license at `http://servet.des.udc.es`.

Acknowledgments

This work was funded by the Galician Government (Xunta de Galicia) under Project INCITE08PXIB105161PR, and by the Ministry of Science and Innovation of Spain under Project TIN2007-67537-C03-02 and FPU grant AP2008-01578. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

[1] R. Clinton Whaley, Antoine Petitet and Jack J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[2] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[3] Markus Püschel, José M. F. Moura, Jeremy Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson and Nicholas Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[4] Kamen Yotov, Xiaoming Li, Gang Ren, María J. Garzarán, David A. Padua, Keshav Pingali and Paul Stodghill. Is Search Really Necessary to Generate High Performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.

[5] Steve Sistare, Rolf van de Vaart and Eugene Loh. Optimization of MPI Collectives on Clusters of Large-Scale SMPs. In *Proc. 12th Supercomputing Conf. (SC'99)*, pages 23–36, Portland, OR, USA, 1999.

[6] Peter Sanders and Jesper L. Träff. The Hierarchical Factor Algorithm for All-to-All Communication. In *Proc. 8th Euro-Par Conf. (Euro-Par'02)*, volume 2400 of *Lecture Notes in Computer Science*, pages 799–804, Padeborn, Germany, 2002.

[7] Vinod Tipparaju, Jarek Nieplocha and Dhabaleswar K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *Proc. 17th Intl. Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 84–93, Nice, France, 2003.

[8] Basilio B. Fraguela, Yevgen Voronenko and Markus Püschel. Automatic Tuning of Discrete Fourier Transforms Driven by Analytical Modeling. In *18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 271–280, Raleigh, NC, USA, 2009.

[9] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proc. 20th Intl. Conf. on Supercomputing (ICS'06)*, pages 353–360, Cairns, Australia, 2006.

[10] Guillaume Mercier and Jérôme Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115, Espoo, Finland, 2009.

[11] Jin Zhang, Jidong Zhai, Wenguang Chen and Weimin Zheng. Process Mapping for Collective Communications. In *Proc. 15th Euro-Par Conf. (Euro-Par'09)*, volume 5704 of *Lecture Notes in Computer Science*, pages 81–92, Delft, The Netherlands, 2009.

[12] Kamen Yotov, Keshav Pingali and Paul Stodghill. X-Ray: A Tool for Automatic Measurement of Hardware Parameters. In *Proc. 2nd Intl. Conf. on the Quantitative Evaluation of Systems (QEST'05)*, pages 168–177, Torino, Italy, 2005.

[13] Kamen Yotov, Keshav Pingali and Paul Stodghill. Automatic Measurement of Memory Hierarchy Parameters. In *Proc. Intl. Conf. on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 181–192, Banff, Canada, 2005.

[14] Alexandre X. Duchateau, Albert Sidelnik, María J. Garzarán and David A. Padua. P-Ray: A Suite of Micro-benchmarks for Multi-core Architectures. In *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, volume 5335 of *Lecture Notes in Computer Science*, pages 187–201, Edmonton, Canada, 2008.

[15] Rafael H. Saavedra and Alan J. Smith. Measuring Cache and TLB Performance and their Effect on Benchmark Runtimes. *IEEE Trans. Computers*, 44(10):1223–1235, 1995.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006.

[17] STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/, Last visit: January 2010.

[18] David E. Culler, Richard Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. 4th Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 1–12, San Diego, CA, USA, 1993.

[19] Roger W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[20] Guillermo L. Taboada, Juan Touriño and Ramón Doallo. Performance Analysis of Message-Passing Libraries on High-Speed Clusters. *Intl. Journal of Computer Systems Science & Engineering*, 2010 (In press).

[21] Finis Terrae. http://www.top500.org/system/9500, Last visit: January 2010.