Load Balancing Using Dynamic Cache Allocation

Miquel Moreto
BSC and DAC, UPC
Barcelona, Spain
mmoreto@ac.upc.edu

Francisco J. Cazorla
IIIA-CSIC and BSC
Barcelona, Spain
francisco.cazorla@bsc.es

Rizos Sakellariou
University of Manchester
Manchester, United Kingdom
rizos@cs.man.ac.uk

Mateo Valero
BSC and DAC, UPC
Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

Supercomputers need a huge budget to be built and maintained. To maximize the usage of their resources, application developers spend time to optimize the code of the parallel applications and minimize execution time. Despite this effort, *load imbalance* still arises in many optimized applications due to causes not controlled by the application developer, resulting in significant performance degradation and waste of CPU time. If the nodes of the supercomputer use chip multiprocessors, this problem may become even worse, as the interaction between different threads inside the chip may affect their performance in an unpredictable way.

Although there are many techniques to address load imbalance at run-time, as it happens, these techniques may not be particularly effective when the cause of the imbalance is due to the performance sensitivity of the parallel threads when accessing a shared cache. To this end, we present a novel run-time mechanism, with minimal hardware, that automatically tries to balance parallel applications using dynamic cache allocation. The mechanism detects which applications may be sensitive to cache allocation and reduces imbalance by assigning more cache space to the slowest threads. The efficiency of our proposed mechanism is demonstrated with both synthetic workloads and a real-world parallel application. In the former case, we reduce the execution time by up to 28.9%; in the latter case, our proposal reduces the imbalance of a non-optimized version of the application to the values obtained with a hand-tuned version of the same application.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*; B.3.2 [**Memory Structures**]: Design Styles—*cache memories*

General Terms

Design, Performance

Keywords

CMP Architectures, Load Balancing, Cache Partitioning

CF'10, May 17–19, 2010, Bertinoro, Italy.

1. INTRODUCTION

In order to obtain good performance from parallel applications it is essential to guarantee that, during execution, the amount of time a processor is waiting for other processors' results is kept at a minimum. Achieving this requires good *load balancing* of the parallel threads. Expert programmers often spend a significant amount of time optimizing work and data distribution in parallel applications so that any potential sources of *load imbalance* are eliminated. However, no matter how good their work is, there may still be issues that cause load imbalance during the application's execution, which may not be easily addressable *a priori*.

According to the classification in [3], there are two main classes of load imbalance due to causes that become apparent only during the application's execution: (i) *Intrinsic load imbalance* is caused by characteristics, which are intrinsic to the application, such as the input data. For example, sparse matrix computations heavily depend on the number of non-zero values in the matrix; the convergence time of iterative methods that approximate the solution of a problem may change for different domains of the modeled space. It is highly difficult (even though not impossible) for the application programmer to balance the application *a priori* to deal with all possible input data sets. (ii) *Extrinsic load imbalance* is caused by external (to the application) factors, that may slow down some processes but not others. A major source of load imbalance could be the operating system (OS) when performing services such as handling interrupts, reclaiming or assigning memory, etc. For example, the OS may decide to run another process (say a kernel daemon) in place of the process running on a CPU. Also, extrinsic load imbalance may be caused by thread contention for processor's shared resources; this may be particularly true in the case of SMT architectures, where threads share and compete for most of the processor's resources [3]. Clearly, there is nothing that the application programmer could do *a priori* to prevent extrinsic load imbalance.

A standard way to address the aforementioned two types of load imbalance is to use dynamic load balancing mechanisms, which are triggered, as necessary, at run-time. We can distinguish between two main strategies to do dynamic load balancing: *work* and/or *data redistribution* and *resources redistribution* [3]. The former strategy includes run-time mechanisms which move some work or data (load) from processes whose execution lags behind to fast-running threads [16, 24]. The latter strategy relates to run-time mechanisms, which may dynamically assign more resources to slow-running threads. Such resources are mainly the number of CPUs,

with more CPUs allocated to slow-running threads [8, 9]. Other authors make use of *hardware thread priorities* to change instruction's decode rate of each thread running on an SMT architecture. This type of software-controllable processor resource allocation allows balancing parallel applications [3, 4], but memory-bound applications are insensitive to decode priority [2] and many CMP processors have only one thread per core.

Adjusting the workload of each thread or adding more processors to slow-running threads may seem a natural solution to the problem. However, uniformingly resorting to such solutions may forfeit the possible benefits stemming from other options, which may sometimes be potentially closer to the root of the problem. For instance, focusing on CMPs, most of them share some cache resources. It has been demonstrated many times in the literature that such a sharing may affect the performance of the threads that share the cache [7, 13, 21, 27]. A corollary of the body of work is that, in CMPs, the cache replacement policy implicitly determines the relative speed of each thread. Generalizing this observation, the hypothesis of this paper is that we can dynamically partition the cache (shared by the parallel threads) in a way that the impact of the cache replacement policy (and its direct impact on the relative speed of each thread) leads to *load balance*. Although there has been a significant amount of work on dynamic cache partitioning this has focused on issues such as fairness, throughput, or ensuring a minimum performance for an application [12, 14, 21, 27]. To the best of our knowledge, this paper is the first work to describe a strategy for dynamic cache partitioning whose objective is to achieve load balance. Conversely, we are not aware of any dynamic load balancing strategies that are based on controlling the cache allocation.

To this end, this paper takes advantage of the opportunity that shared caches in CMP architectures offer to propose a dynamic mechanism to reduce the load imbalance of parallel applications (whose threads share a CMP's cache). Our mechanism detects in which situations cache allocation can be used to balance applications and, in these situations, it assigns more cache resources to processes computing longer. The iterative nature of many parallel applications facilitates this task because the behavior of an application in previous iterations can be used as a *learning phase* of the algorithm in the following iterations. Furthermore, we analyze the time granularity at which cache allocation decisions should be taken. We explore both application granularity at iteration-level (in the order of milliseconds), in which the balancing algorithm can be executed in software, and a finer granularity that requires minimal hardware support. We conclude that the software solution provides better results as it has a global vision of the imbalance of the application.

When applied to synthetic workloads, our suggested mechanism can reduce execution time by up to 28.9%. When applied to a real-world parallel application, `wrf` [17], our mechanism helps reduce the expensive and long optimization time that expert programmers spend hand-tuning the application. In particular, our proposal reduces the load imbalance of a non-optimized version of `wrf` to values comparable to an optimized version of `wrf`, in which several man-years of effort have been devoted to balance the application; the overall reduction in execution time achieved is about 7.4%. This suggests that our balancing mechanism may represent an alternative approach to reduce considerably the development time invested in balancing parallel applications manually.

The rest of this paper is structured as follows. Section 2 presents the motivation for this work. The dynamic mechanisms to balance applications are explained in detail in Section 3. Section 4 describes the experimental environment, while, in Section 5, simulation results for synthetic workloads are discussed. Section 6 evaluates the proposed mechanisms with a real-world parallel application. Finally, Section 7 concludes the paper.

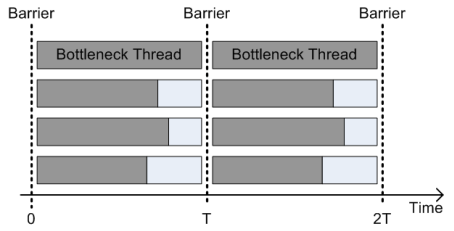
2. MOTIVATION

In CMP architectures, other shared resources like cache space can be re-distributed to balance parallel applications. Caches are built up with equally-sized groups of cache lines, called *sets*. The size of each cache set is called the K-associativity of the cache. Different cache lines that collide into the same cache set can be distributed along the K different ways of the set. When a new request arrives to a set that is full, a cache eviction policy, like LRU, chooses the victim line to evict from the cache set. The LRU eviction policy is demand-driven and tends to give more cache space to threads accessing more frequently the cache hierarchy. Moreover, the OS and software cannot exercise any control over how threads share a cache when using LRU as eviction policy. In contrast, *way-partitioned* caches provide the opportunity to control cache allocation from the software level, which can lead to significant performance speed ups for sequential applications [14,21,27]. A way-partitioned cache prevents threads from negative interference: a thread checks all cache lines in a set when it accesses the cache, but is allowed only to evict lines from a reduced number of ways [27]. This ensures correctness without needing to flush the data from one way (when a thread relinquishes it) or to share a virtual address space among threads. Partitioned caches can also be used for ensuring a minimum performance to applications [12].

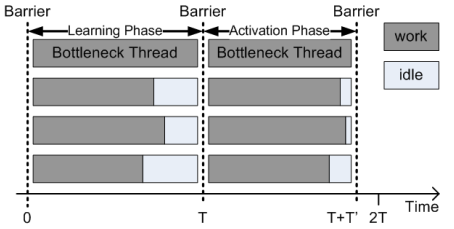
The use of cache allocation control mechanisms may not be a useful load balancing mechanism in all execution settings. In some cases, the execution of the parallel threads may be highly cache sensitive (and, hence, it would be feasible to balance the load dynamically through appropriate adjustment of cache resources), but, in other cases, the parallel threads may be cache insensitive (in which case, different ways to allocate cache resources may have little or no impact on their performance). Detecting whether a particular parallel workload is cache sensitive or not is something that cannot be assumed to be known; instead, an on-the-fly assessment of the parallel workload, to *learn* whether it is cache sensitive or not, would be required. Assuming that the parallel workload follows some kind of an iterative pattern, where there is a repetition of a sequence consisting of a computation phase followed by a barrier synchronization, then the behaviour of the application in the previous *iteration* can be used to make decisions for the next *iteration* [1].

To illustrate this, consider the example shown in Figure 1(a), which shows the execution of a parallel application running with four threads. The application consists

[1]Iterative patterns are often encountered in many High-Performance Computing (HPC) Applications. For example, a partial differential equation (PDE) solver may consist of an outermost sequential loop inside which a parallel loop is executed. This specifies a number of iterations each of which completes with a call at a barrier at the joint point of the parallel loop that synchronizes all threads.



(a) Execution without controlling the shared LLC



(b) Execution after assigning more cache space to the bottleneck thread

Figure 1: Synthetic example of a parallel application with 4 threads running in the same CMP.

of a number of iterations, in each of which a computation phase is followed by barrier synchronization; two iterations are shown in the figure. The first thread in Figure 1(a) is the bottleneck (*bottleneck thread*), as other threads have to wait at the barrier until it finishes executing. Assuming that the threads run in a multithreaded processor sharing the last level cache (LLC), ideally, assigning more cache resources to the first thread would reduce its execution time without excessively degrading the other threads' performance. Figure 1(b) shows the execution of the application after a new assignment of the cache resources has been applied to the second iteration; it is assumed that $T' < T$. If the parallel application consists of many more iterations, the new cache allocation can be used to reduce the overall execution time of the application. The idea is that, in one iteration, an appropriate detection mechanism learns the behavior of the application (*learning phase*) and activates a balancing algorithm from the next iteration (*activation phase*).

Anecdotal evidence collected after many man-years of experience with several HPC applications running on the MareNostrum supercomputer suggests that, usually, intrinsic load imbalance in HPC applications is due to two main reasons. First, all processes of the HPC application execute the same number of instructions but each of them has a different LLC behavior, as some processes experience more LLC misses than others. Second, all processes of the HPC application have a similar LLC behavior but the number of instructions each of them executes is different.

As a matter of a real-world example, Figure 2 shows part of the execution of a parallel MPI application, `wrf` (which will be explained in detail in Section 6), when running on a supercomputer with 64 MPI processes. Here we show the execution of the first 16 threads for simplicity; the same behavior is observed in the other 48 MPI processes. Each application iteration takes about 1.5 seconds; one iteration is shown in Figure 2. The iteration is comprised by several *computation phases*, denoted C_i, each having a different color. At the end of each computation phase, a synchronization phase begins. Communication and waiting time are marked in white. The iteration begins with a long computation phase of about 0.3 seconds denoted C_1. Note that many

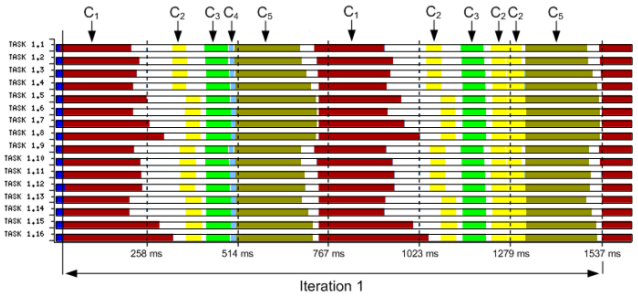


Figure 2: Execution of the `wrf` HPC application with 64 threads. Only the first 16 are shown for simplicity. The same behavior is observed in the other 48 MPI processes

MPI processes have to wait for a long period of time until all threads finish this phase and reach the barrier. This computation phase is a clear representative of the second scenario of intrinsic imbalance mentioned before (where the number of instructions of each thread is different), as process 14 executes 40.1% less instructions than process 16. Next, there are three short phases (C_2, C_3 and C_4, during 0.2 seconds in total) and then, a long computation phase begins (C_5 between times 0.5 and 0.75 seconds). This phase is more balanced than the first one, but load imbalance is still present. In phases C_2, C_3 and C_4 the number of executed instructions is the same and the load imbalance is mainly due to a different LLC behavior. Afterwards, in the same application iteration, the execution continues with computation phases C_1, C_2, C_3, C_2 and C_5. As will be shown in our experimental results section, adjusting cache resources allocated to each thread, using an appropriate learning mechanism in one iteration and applying it to the next iteration, allowed us to achieve a performance improvement for the application, which is comparable to the performance obtained through manual tuning after several man-years of effort.

A key idea in this example is the notion of an *iteration*. Detecting iterations in a parallel application can be done with different approaches. Static off-line profiling information can be used to inform the OS about the iteration borders. Some authors propose the introduction of checkpoints in the source code of the parallel application to identify possible unbalanced points in a parallel loop [5]. Other authors make use of run-time libraries that dynamically measure the percentage of load imbalance per process [8]. In fact, parallel applications alternate *computing phases* with *waiting phases* (when a process is waiting for synchronization). Thus, some authors consider the sum of a computing and a waiting phase as one iteration of the parallel application [4]. Other solutions based on analyzing performance counters are also useful to detect these parallel iterations [6]. On-line detection of iterations is still an open research issue that is beyond the scope of this paper.

3. DYNAMIC LOAD BALANCING THROUGH CACHE ALLOCATION

In this section, we describe in detail two different algorithms, which implement a mechanism that can change dynamically the cache allocation of a parallel application, as it runs, to reduce load imbalance. It is assumed that the parallel application consists of *iterations*, as described above. The key idea is that at the end of each iteration our mechanism invokes one of the two proposed algorithms to make a decision on whether to change the current cache allocation or not. The decision of the algorithms is based on an anal-

ysis of the behavior of the parallel threads in the preceding iteration; this decision is then applied to the immediately following iteration. The main difference of the two algorithms is their complexity. Thus, the first algorithm implements a simple heuristic that tries to give more cache space (one extra way in the cache) to slow threads by removing an equal amount of cache space from fast threads, while the second algorithm tries to (re-)allocate cache space for all threads at once, in a way that minimizes their overall execution time. The two algorithms are presented next.

3.1 Iterative Method: Load Imbalance Minimization (MinLoadImb)

The first algorithm proposed is based on monitoring the execution time of running threads at each iteration. Then, at the end of the iteration, the degree of load imbalance is calculated and the algorithm to reallocate ways of the cache is invoked if this imbalance is above a threshold ϵ_1.

To measure the degree of load imbalance amongst parallel threads two metrics have been suggested: the *relative load imbalance* [23] and the *imbalance percentage* (IP) [22]. The former is a ratio of the deviation of the execution time of the longest running thread from the average execution time of the threads divided by the execution time of the longest running thread; the latter is a normalized version of the former with values between 0 and 100. High values indicate high load imbalance. We chose the latter because it makes understanding easier especially when dealing with small numbers of threads. Thus, if we have a parallel application with N processes, $N \geq 2$, we define

$$IP = 100 \cdot \frac{MaxExecTime - AverageExecTime}{MaxExecTime} \cdot \frac{N}{N-1}.$$

Intuitively, it is useful to see the imbalance percentage (IP) as the average percentage of time that the parallel threads are waiting at the end of a parallel section for the slowest thread to finish [22].

Algorithm 1: An iterative method for load balancing using dynamic cache allocation: MinLoadImb

```
Data: Threshold values 0 ≤ ε1, ε2 ≤ 1, execution time of all
      threads in the previous iteration, ET_i^previous, and
      current eviction policy (LRU or partitioned cache)
Result: Final cache allocation for the next iteration (LRU or
        new cache partition)
begin
    compute imbalance percentage, IP
    //activation mechanism
    if (LRU is the current eviction policy) then
        if (IP/100 < ε1) then choose LRU and stop
        else choose a partitioned cache.
    end
    L ← list of threads sorted by ET_i^previous
    while number of threads in L ≥ 2 do
        remove the slowest thread s from L
        find_fastest_thread:
            remove the fastest thread f from L
            if (thread f has no more than one assigned way)
              then if (L is not empty)
                       then goto find_fastest_thread
                       else stop
        if ((ET_s^previous − ET_f^previous) / ET_f^previous > ε2) then assign one
        way from thread f to thread s
    end
    stop
end
```

Once the balancing algorithm is activated, at the end of each iteration of the parallel application, it tries to remove one way (from the cache allocation) of the fastest threads and assign it to the slowest threads. To do this, threads are ordered in terms of their execution time in the previous iteration, $ET_i^{previous}$. Then, the fastest thread surrenders one way to the slowest thread, then the second fastest to the second slowest and so on. This is repeated until the ratio of the next slowest thread to the next fastest thread does not exceed the value of a given threshold ϵ_2 plus 1, or there are no more threads to consider. During this process, the algorithm ensures that all threads retain at least one way from the cache allocation.

The intuition behind this proposal is that taking (cache) resources from fastest threads and allocating them to slowest threads may hopefully reduce the overall load imbalance. The proposed algorithm is denoted by *MinLoadImb* and is described in Algorithm 1. Note that the cache partition is not changed during the execution of an iteration of the application but only when the iteration completes. The algorithm uses as an input the execution time of each thread in the previous iteration $ET_i^{previous}$, which can be obtained using performance counters. The algorithm makes also use of two thresholds, ϵ_1 and ϵ_2. The value of ϵ_1 controls how often the algorithm will be invoked. High values indicate that the algorithm will be rarely invoked, only in cases of a high imbalance percentage. Conversely, the value of ϵ_2 controls how far the reallocation of ways from the fastest to the slowest threads would go. Again, high values indicate that only a small number of the fastest and slowest threads will be considered for reallocation of ways. After performing a sensitivity study, we concluded that good performance is obtained when $\epsilon_1 = 0.075$ and $\epsilon_2 = 0.025$.

3.2 Single-step Method: Execution Time Minimization (MinExecTime)

The second algorithm proposed is denoted *MinExecTime*. Through specialized hardware [19], *MinExecTime* can estimate the execution time of a given parallel thread with a different cache assignment (for example, when a specific number of ways are assigned to it). This information can be used to find an optimal cache partition. Same as before, This algorithm is also invoked at the end of each iteration of the parallel application.

To understand how to use the information provided by the specialized hardware we describe the problem to be solved next. We define $\phi(\bar{k}) = \max_i\{ET_i(k_i)\}$ the execution time of the application with the cache partition $\bar{k} = (k_1, \ldots, k_N)$, where $ET_i(k_i)$ is the execution time of thread i when k_i ways are assigned to it. Then, we are looking for the optimal cache partition $\bar{k}_{opt}$, which is the one that minimizes the execution time of the application, that is:

$$\phi(\bar{k}_{opt}) = \min_{\bar{k}=(k_1,\ldots,k_N)} \left\{ \phi(\bar{k}) \;\middle|\; \sum_{i=1}^{N} k_i = K \right\}$$

Checking all possible combinations for the values of k_i would be too expensive. However, we can avoid doing this by noting that $ET_i(k_i)$ is monotonically non-increasing and that $\phi(\bar{k})$ is determined by the execution time of the slowest thread. This means that we can find an optimal cache partition by following a procedure that: (i) starts with the assignment of one way to each thread; and, (ii) assigns the

Algorithm 2: A single-step method for load balancing using dynamic cache allocation: MinExecTime

Data: Threshold values $0 \leq \theta \leq 1$ and execution time of the previous iteration with the previous cache allocation, $ET_{previous}$
Result: Final cache allocation for the next iteration (LRU or new cache partition)

```
begin
    assign one way to each thread
    estimate ET_i for each thread with the current cache allocation
    while (available ways exist) do
        find the slowest thread s
        assign one extra way to thread s
        estimate ET_s
    end
    find the slowest thread s
    ET_MinExecTime ← ET_s
    //activation mechanism
    if ((ET_previous − ET_MinExecTime) / ET_previous > θ)
        then apply the new cache partition found above
        else keep the previous cache allocation (LRU or previous cache partition)
    stop
end
```

remaining ways, one-by-one, to the slowest thread (which thread is the slowest is recalculated after the assignment of each additional way according to the hardware estimates)

The algorithm is illustrated in Algorithm 2. Initially, one way is assigned to every thread. To minimize load imbalance, we need to minimize the execution time of the slowest thread. Clearly, it is not possible to do this if we do not assign more cache resources to this particular thread. Thus, the algorithm assigns, in each step, one extra way to the estimated slowest thread. Then, the algorithm obtains the optimal solution in $K - N$ iterations of the while loop, where K is the cache associativity and N is the number of threads. Finally, if the optimal cache partition is an improvement over LRU by more than a threshold value θ, this partition is applied to the next iteration of the application. The algorithm uses as an input the execution time of the application in the previous iteration $ET_{previous}$, which can be obtained with current performance counters. Also, with respect to the threshold θ, following sensitivity analysis with exhaustive simulations, we chose to use in our experiments a value of $\theta = 0.05$.

A critical aspect of this algorithm is the use of specialized hardware to obtain performance estimations. We have evaluated different approaches to obtain such estimations [19, 28] and concluded that having better accuracy in performance estimations leads to larger speed ups and less wrong activation decisions. The best results are obtained using *OPACU* [19], as it has an average 3.11% error over the whole SPEC CPU 2000 [26] benchmark suite. In Section 5 we show that thanks to the high accuracy of this mechanism, we never activate the cache partitioning mechanism in a situation where it would worsen performance.

In a CMP architecture with a shared L2 cache that is the last level cache on-chip, OPACU uses a sampled Auxiliary Tag Directory (ATD) to obtain the number of misses per L2 configuration as in [14, 21, 27]. The ATD has the same associativity and size as the tag directory of the shared L2 cache and uses the same replacement policy. It stores the behavior of memory accesses per thread in isolation. While the tag directory of the L2 cache is accessed by all threads, the ATD

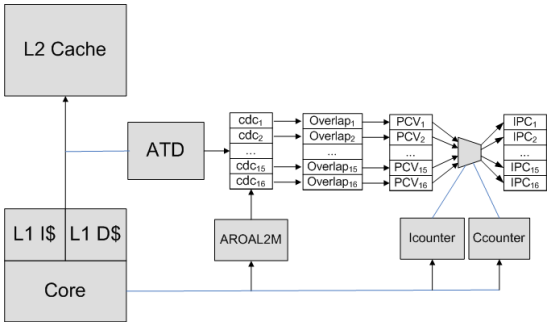


Figure 3: OPACU's hardware implementation for one core with a 16-way L2 cache

of a given thread is only accessed by the memory operations of that particular thread. In out-of-order architectures, different cache misses can occur concurrently if they all fit in the Reorder Buffer (ROB), which allows exploiting the Memory Level Parallelism (MLP) of the application. A group of cache misses is denoted a *cluster* of misses. OPACU uses a reduced number of hardware counters to determine if L2 data misses are clustered or not. Three counters per core are needed: number of instructions, cycles, and average ROB usage after an L2 miss ($AROAL2M$). Three counters per cache assignment are also needed: last L2 miss identifier (cdc_i), number of clusters ($overlap_i$) and predicted total waiting cycles due to an L2 miss (PCV_i). When a load instruction accesses the L2, the sampled ATD is used to determine if it would be a miss in other possible cache assignments. Using the last L2 miss identifier, it can be determined whether a new cluster of misses begins or not. The number of clusters and lost cycles is updated accordingly. With this information, we can predict the IPC of the process under different cache configurations. Figure 3 illustrates this mechanism.

3.3 Comparison of the Algorithms

Next, we list the main differences between the introduced balancing algorithms.

Activation. The activation decision represents a key feature of the load balancing mechanism. Wrong activations may lead to performance degradation as not all applications benefit from cache partition mechanisms: at one extreme, one application suffers a 14% slowdown in our experiments when the balancing mechanisms are activated blindly. *MinLoadImb* estimates the load imbalance with the execution time of each thread in the previous iteration. However, this proposal is not aware of the effect that the assignment of one extra way to a thread will have on the performance of the other threads. As a consequence, *MinLoadImb* may suffer a performance degradation due to wrong activation decisions. Instead, *MinExecTime* is based on direct execution time estimations. In Section 5 we show that this mechanism has a high accuracy and that it never activates the dynamic cache partitioning mechanism in a situation where performance degradation would have been obtained.

Convergence time. The time needed to converge to the optimal solution is different for each mechanism. *MinLoadImb* assigns at most one extra way per thread every time it is called, while *MinExecTime* has no limit in new cache assignments. Thus, *MinExecTime* will respond quicker to the imbalance of an application and reach sooner the optimal partition.

Hardware Cost. The hardware cost of these mechanisms is significantly different. *MinLoadImb* decides cache

partitions with nearly no hardware overhead, as it can read performance counters to obtain the execution time of each thread in the previous iteration. Instead, *MinExecTime* requires special hardware to obtain performance estimations for all cache configurations. For a 4-core CMP with a shared 1MB 16-way L2 cache, OPACU needs less than 1KB of total storage per core (including a sampled ATD [21] and all the required hardware counters [19]). Some authors have embedded the monitoring logic inside the L2 cache, devoting some sets to monitor each thread [13]. Using a similar approach, the hardware cost of OPACU would be reduced to 204 bytes per core with a 16-way L2 cache.

4. EXPERIMENTAL SETUP

Simulation Framework. In this paper we use MPsim [1], a flexible cycle-accurate simulator that allows us to model CMP architectures. Each core in the CMP has its own data and instruction L1 caches, while the unified L2 cache is shared among cores in the CMP. The L2 is the LLC on-chip. Each core is single threaded and can fetch up to 8 instructions each cycle. The decode bandwidth is 5 instructions per cycle. It has 6 integer (I), 3 floating point (FP), and 4 load/store functional units and 32-entry I, load/store, and FP instruction queues. Each thread has its own 256-entry reorder buffer and 256 physical registers. We use a two-level cache hierarchy with 128B lines with separate 16KB, 2-way associative data cache and a 32KB direct-mapped instruction cache, and a unified 1MB, 16-way L2 cache that is shared among all cores. The cache configuration evaluated in this paper is similar to the one currently working in our supercomputer infrastructure. Latency from L1 to L2 is 15 cycles, and from L2 to memory 300 cycles. We use a 32B width bus to access L2 and a multibanked L2 of 16 banks with 3 cycles of access time. We model bus and banks conflicts, and a single channel to access main memory.

Workloads. To evaluate our dynamic load balancing algorithms, presented in Section 3, we use both synthetic workloads and a real-world parallel application in production at our supercomputing center. Synthetic workloads allow us to evaluate our proposals in a wide range of scenarios, while the real-world HPC application completes the study and demonstrates the robustness of our proposals.

As pointed out in Section 2, there are two main situations of intrinsic load imbalance: when all threads execute the same number of instructions but have different cache behavior, and when all threads have a similar cache behavior but the number of executed instructions is different. The objective of the synthetic workloads is to mimic these situations and show that our proposals reduce load imbalance in both situations. We compose 2-thread workloads from SPEC CPU 2000 [26] benchmarks. From each benchmark we collected traces of the most representative 300 million instruction segment of each program, following the SimPoint methodology [25]. We artificially add a barrier at the beginning and end of the execution of these benchmarks to mimic an HPC application. In our simulation methodology, each thread introduces a different skew to virtual addresses, which avoids that different cores hit the same cache set. Furthermore, each copy of the same program is forwarded a different number of instructions to avoid that each copy is exactly at the same point of execution.

These synthetic workloads are designed to have different cache requirements depending on their *building* benchmarks.

Table 1: Benchmark classification in groups L, S and H. For each benchmark, we report the number of accesses to the L2 cache per thousand cycles (APTC) and the IPC when running in isolation

L Benchmarks	APTC	IPC	**H** Benchmarks	APTC	IPC
applu	16.83	1.03	ammp	23.63	1.27
bzip2	1.18	2.62	apsi	21.14	2.17
equake	18.6	0.27	art	46.04	0.52
gap	2.68	0.96	facerec	10.96	1.16
lucas	7.60	0.35	fma3d	15.1	0.11
mcf	9.12	0.06	galgel	18.9	1.14
mesa	3.98	3.04	mgrid	9.52	0.71
sixtrack	1.34	2.02	parser	9.09	0.89
swim	28.0	0.40	twolf	12.0	0.81
wupwise	5.99	1.32	vpr	11.9	0.97
S Benchmarks	APTC	IPC	**S** Benchmarks	APTC	IPC
crafty	7.66	1.71	eon	7.09	2.31
gcc	6.97	1.64	gzip	21.5	2.20
perl	3.82	2.68	vortex	9.65	1.35

To that end, we classify SPEC CPU 2000 benchmarks into three groups: *Low utility* (L), *Small working set* or *saturated utility* (S) and *High utility* (H) [18, 21]. Table 1 lists the benchmarks belonging to these three groups. We also show the average number of L2 accesses per thousand cycles (APTC) and the IPC for each benchmark.

Finally, we evaluate our load balancing algorithms with real traces from a parallel application running on an actual supercomputer: `wrf`. We used two versions of this application: (i) an *optimized* version, currently in production at our supercomputing center, in which several man-years of effort have been devoted to its (manual) optimization; and (ii) a *non-optimized* version of the same application in which less optimization techniques have been applied. Thus, we have two versions of the same application, `wrf`, with different imbalance percentages. Section 6 presents our evaluation results, including a full description of how traces of `wrf` were obtained, which was a complex and time-consuming task.

5. PERFORMANCE CHARACTERIZATION WITH SYNTHETIC WORKLOADS

In this section we analyze the behavior of the load balancing algorithms, presented in Section 3, using synthetic workloads. In the experiments in this section, we run two benchmarks simultaneously. When the fastest benchmark ends, the slowest keeps executing until it finishes its execution. This is what we call an *iteration*. In each execution we run ten iterations. It is only at the end of the first iteration that the load balancing algorithms may be activated (since the first iteration is used purely as a learning phase), thus, we report the average execution time of the final nine iterations with respect to the results of the first iteration.

As pointed out in Section 2, there are two situations of (intrinsic) load imbalance that can be solved with our proposal. Section 5.1 evaluates the situation in which all threads execute a similar number of instructions but have different L2 cache behavior. Next, Section 5.2 evaluates the situation where all threads have a similar L2 cache behavior but the number of instructions is different.

5.1 Load Imbalance due to Different L2 Cache Behavior

To mimic the situation where all threads execute the same number of instructions but have different L2 cache behavior, we randomly generate 36 pairs of benchmarks with different cache requirements. We choose the workloads listed in Ta-

Table 2: Workloads of benchmarks with different L2 cache behavior. Four workloads per group are chosen

	Slow Th	Fast Th	IP		Slow Th	Fast Th	IP
LL1	sixtrack	bzip2	44.8	LS1	applu	gzip	48.3
LL2	mcf	wupwise	95.2	LS2	sixtrack	eon	33.9
LL3	lucas	gap	63.5	LS3	mcf	vortex	95.3
LL4	equake	swim	30.8	LS4	gap	perlbmk	62.4
LH1	lucas	art	29.9	SL1	perlbmk	mesa	5.95
LH2	swim	vpr	9.19	SL2	vortex	wupwise	2.11
LH3	mcf	facerec	93.6	SL3	gcc	sixtrack	8.78
LH4	wupwise	apsi	35.7	SL4	crafty	bzip2	37.0
SS1	crafty	eon	23.3	SH1	crafty	apsi	19.3
SS2	vortex	crafty	14.8	SH2	gcc	apsi	20.6
SS3	gcc	perlbmk	35.1	SH3	vortex	ammp	16.2
SS4	vortex	gzip	40.8	SH4	vortex	apsi	33.6
HL1	galgel	bzip2	60.8	HS1	twolf	crafty	54.0
HL2	mgrid	gap	28.5	HS2	ammp	gcc	30.4
HL3	parser	applu	30.9	HS3	gap	mgrid	28.5
HL4	fma3d	swim	73.8	HS4	parser	perlbmk	65.8
HH1	galgel	ammp	6.42	HH2	parser	apsi	60.1
HH3	twolf	art	16.6	HH4	vpr	facerec	26.8

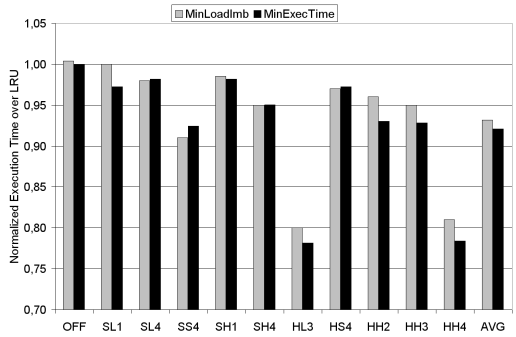
ble 2. We denote XY the pairings of benchmarks where X and Y are the slowest and fastest thread, respectively.

Figure 4(a) shows the total execution time for both load balancing algorithms: *MinLoadImb* and *MinExecTime*. All results are normalized to the execution time of the first iteration, in which the LRU replacement policy is used. All workloads in which *MinExecTime* decides to make use of LRU as eviction policy are represented in the first set of bars, denoted *OFF* (as the cache partitioning algorithm is not activated) with their average results. Clearly, the average normalized execution time with respect to LRU is equal to 1 in case of *MinExecTime*. *MinLoadImb* is sometimes triggered in these workloads, without leading to any improvement of the execution time with respect to LRU. In fact, the imbalance percentage is reduced from 40% to 20% but this is not translated into any reduction in the execution time. This result suggests that the activation mechanism of *MinExecTime* is more accurate than the one of *MinLoadImb*.

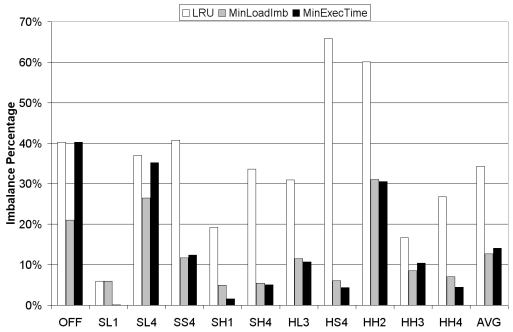
All algorithms present similar results in groups LL, LS and LH as the shared L2 has little impact on the performance of the bottleneck thread. Consequently, *MinExecTime* detects that there is no room for improvement and keeps using LRU as eviction policy (the results are represented in bar *OFF*). In contrast, *MinLoadImb* is triggered in all these workloads as the imbalance percentage is always greater than 7.5%.

When LRU is not harming the performance of the slowest thread, there is no room for improvement. The *MinExecTime* activation mechanism detects these situations and does not try to balance the application. Instead, *MinLoadImb* is triggered in 89.6% of the situations without significantly reducing execution time. In general, *MinLoadImb* is triggered more often than *MinExecTime*, even in situations where significant performance degradations are obtained (9.6% in SH3 and 8.4% in HS2[2]). When LRU is clearly biased toward the fastest thread (SS4, HL3, HH2, HH3 and HH4), the reduction in execution time is more significant, ranging between 4.1% and 21.6% (HH4). In intermediate situations, the bottleneck thread is less affected by the fast thread and the performance speed ups are smaller (between 1.5% and 4.5%). In some situations, suboptimal partitions in terms of execution time manage to reduce even more the load imbalance of the application. In these situations (like HH3), all threads are slower but are more balanced. This observation justifies that *MinExecTime* leads to better partitions than *MinLoadImb* as it is using direct estimations of the execution time instead of the IP metric.

[2]Workloads SH3 and HS2 are not on Figures 4(a) and 4(b) as they are represented in the bar *OFF*



(a) Execution time normalized over LRU



(b) Imbalance percentage

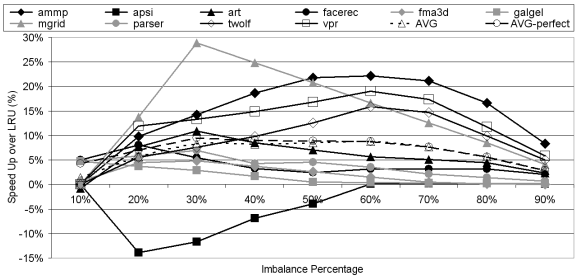
Figure 4: Execution time and imbalance percentage for pairings with different L2 cache behavior

MinLoadImb converges more slowly to the optimal solution than *MinExecTime*, so it normally returns worse results. However, as the number of total iterations of the application increases, this difference in performance decreases as both algorithms eventually reach an optimal partition. In some pairings *MinLoadImb* gets stuck in a local minimum because the threshold ϵ_2 may be too big for the situation, while in others it suffers a ping-pong effect between the optimal and a suboptimal partition (because ϵ_2 may be too small for the situation). These problems do not occur with *MinExecTime* as it estimates the execution time for each cache configuration and finds out the optimal partition in one step.

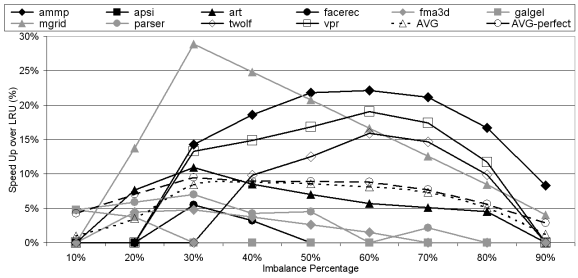
On average, *MinLoadImb* and *MinExecTime* outperform LRU by 6.9% and 7.9%, respectively. Figure 4(b) shows the imbalance percentage for both algorithms with different workloads. Our algorithms reduce the imbalance percentage from an overall average of 34.3% to 12.7% (*MinLoadImb*) and 14.1% (*MinExecTime*). In general, *MinLoadImb* succeeds in reducing more the imbalance percentage than *MinExecTime* as this metric (that is, load imbalance minimization) is guiding this algorithm.

5.2 Load Imbalance due to a Different Instruction Count

In the next experiment, we run two instances of the *same* benchmark and we artificially generate load imbalance by reducing the number of executed instructions of the second thread. To generate an I% of imbalance, we appropriately reduce the number of instructions of the second trace. This experiment mimics the situation where all threads have the same cache behavior but different instruction count. Re-



(a) Speed up over LRU with *MinLoadImb*



(b) Speed up over LRU with *MinExecTime*

Figure 5: Speed-up over LRU for HH pairings of SPEC CPU 2000 benchmarks with an imbalance due to a different number of executed instructions

member that each thread introduces a different skew to virtual addresses so that data fetched by one thread is not used by the other threads. As in the previous section, we repeat ten times each iteration and report the average execution time of the final nine iterations with respect to the results of the first iteration.

Given that both threads have the same cache behavior, LRU devotes a roughly equal portion of the cache to each thread. As a consequence, LL and SS pairings will leave no opportunity for execution time reduction. When the bottleneck thread belongs to group L, it will not be affected by the other executing threads. The same situation will happen if the bottleneck thread belongs to group S, as the cache requirements of each thread are fulfilled. Thus, for these benchmarks we cannot obtain significant improvements in execution time, even for high values of imbalance percentage. In these situations, *MinExecTime* is able to detect that it cannot obtain any performance gain, and it keeps LRU as replacement policy avoiding any performance degradation. Instead, *MinLoadImb* is triggered in 95% of the situations with a negligigle impact on the execution time. For HH pairings, the situation is different as an execution time reduction is possible through cache partitioning. Thus, we study the pairs consisting of two instances of each of the 10 H benchmarks in more detail next.

The speed-up that both proposed algorithms achieve over LRU with the 10 pairs of the H benchmarks and for an imbalance percentage varying from 10% to 90% is shown in Figures 5(a) and 5(b). The line denoted with *AVG* indicates the average results with each algorithm among all benchmarks for a given imbalance percentage. In contrast, the line denoted with *AVG-perfect* indicates the average results of a partitioning algorithm with perfect knowledge that always makes the correct decision.

Studying the behavior of *MinLoadImb* in Figure 5(a), we observe that all pairs except `apsi` obtain significant performance speed ups as we vary the imbalance percentage of the application. In the case of `apsi` there is a performance degradation of 13.9% with an imbalance percentage of 20%. This is a consequence of the nature of *MinLoadImb* and its activation mechanism. In contrast, *MinExecTime* never degrades performance with respect to LRU. This is corroborated by the results in Figure 5(b). In fact, the activation mechanism of *MinExecTime* is conservative and may lose opportunities for performance improvements with small imbalance percentages. However, this is the cost that has to be paid in order to never lose performance with respect to LRU. Thus, on average, the speed up achieved by *MinExecTime* is 5.8%. In contrast, the *MinLoadImb* activation mechanism is much more aggressive and is triggered more often, reaching an average 6.3% improvement. As a consequence, in some situations it loses performance with respect to LRU (up to 13.9% of degradation in case of `apsi`). Finally, it is noted that a perfect prediction mechanism that always tries to balance an application when there are opportunities to do so would obtain an average speed up of 7.1%. The gap is not too high, but reducing it is part of our future work.

Another observation from both figures is that the improvement is, in general, less when the imbalance percentage is very high. This can easily be explained because when the imbalance percentage is, say, 90% it means that most of the time one thread is executing on its own, hence there are only limited opportunities for dynamic cache partitioning. Also, despite the differences of the two algorithms, the behavior of each benchmark is largely determined by its characteristics. Thus, the maximum speed up is obtained with `mgrid` with an imbalance percentage of 30%, reaching a speed up of 28.9%. The behavior of `ammp`, `twolf` and `vpr` corresponds to benchmarks that are very sensitive to cache allocation, reaching speed ups between 15% and 25%. Other benchmarks like `art`, `facerec`, `fma3d`, `galgel`, `mgrid` or `parser` are less sensitive to the cache allocation and exhibit speed ups between 5% and 10%.

Table 3: Accuracy of the activation mechanism with HH pairings

		MinLoadImb decision		*MinExecTime* decision	
		Yes	No	Yes	No
Oracle	Yes (77.7%)	73.3%	4.4%	53.3%	24.4%
	No (22.3%)	21.1%	1.2%	0%	22.3%

Table 3 shows the accuracy of the activation mechanism for *MinLoadImb* and *MinExecTime*. We compare their results with the correct decision that would be taken by an ideal activation mechanism with perfect knowledge. When LRU obtains better performance than a partitioned cache, the correct decision should be to keep LRU as eviction policy (denoted *No* in the 2nd row of Table 3). Conversely, if the partitioned cache obtains better performance, the correct decision should be to activate the balancing algorithm (denoted *Yes* in the 1st row of Table 3). *MinExecTime* is never triggered when there is no opportunity for execution time reduction. This was the main design goal of this mechanism because this situation may lead to heavy execution time degradation of the application. On the other hand, in 53.3% of the cases in which it is possible to benefit from cache partitioning to balance the workload, *MinExectime* is triggered. The lost opportunities are 24.4% of the total cases. As a future work we plan to develop a more aggressive prediction mechanism to decrease this loss of opportunities. Instead, *MinLoadImb* is much more aggressive than *MinExecTime* and is triggered is 94.4% of the situations, leading to better average results, but suffering significant performance degradations in some situations.

5.3 Granularity Analysis of the Load Balancing Mechanism

So far we have assumed that the cache allocation is changed at the boundary of a computation phase (that is, at the end of each iteration). This is a coarse enough granularity to implement this solution in software at the OS or runtime level. The cache partition found out at the end of one iteration is maintained throughout the whole iteration that follows.

As an alternative to the previous approach, we can use a dynamic algorithm that changes the cache partition during the execution of a computation phase. This mechanism continuously monitors the load imbalance (or predicted execution time) and adapts the cache partition to IPC variations inside the same computation phase. An algorithm working at this granularity is invoked periodically and executed in a dedicated hardware, like in [21]. In our baseline configuration cache partitions are decided every 5 million cycles[3]. As cache partitions are modified during the execution of a phase, we denote our algorithms *Dynamic-MinLoadImb* and *Dynamic-MinExecTime*.

Dynamic-MinLoadImb. At the end of the first iteration of the application, the mechanism decides to activate the balancing algorithm if the load imbalance of the application is above a threshold ϵ_1 (implemented at software level). During the next iteration new cache partitions are decided at intervals of 5 million cycles.

This hardware estimates the execution time of each thread with the current cache partition using the current IPC of the application and stores the result in the *Execution Time Table* (ETT). This structure sorts all threads according to their execution time. Once we have filled the ETT with all the values, we use Algorithm 1 to partition the cache: The fastest threads give one extra way to the slowest ones if their difference in execution time is greater than ϵ_2 and the slowest thread has more than one way. No extra ways will be assigned to a thread that has already finished executing. The main difference with the static algorithm is that here we use the predicted remaining execution time instead of the execution time in the last interval. In fact, using the IPC of the application in the last iteration (instead of the current IPC), we will obtain the same cache partition as with *MinLoadImb*.

Dynamic-MinExecTime. The hardware implementation of this proposal is similar to the previous mechanism as it also requires the ETT. The basic difference is that a dedicated hardware [19] provides the IPC of the application with a different cache assignment. First, the execution time of each thread with only one way is estimated and sorted in the ETT. Then, one way is assigned to the slowest thread and its new execution time is estimated and inserted again in the ETT. This process is repeated $K - N$ times until all ways have been assigned (see Algorithm 2).

Next, we account for the total extra storage needed for the versions of the balancing algorithms implemented in hardware explained in this section. In case of *Dynamic-MinLoadImb*, for an 8-core CMP, we need less than 20 bytes of total storage per thread. In case of *Dynamic-MinExecTime*, we need extra storage for the IPC predictions. Thus, for an 8-core CMP with a 16-way L2 cache, we need less than 80 bytes of total storage per thread.

[3]The frequency of cache partition decision has been chosen after evaluating the mechanism for a wide range of values

We have evaluated the dynamic approaches with the same workloads as in the previous sections. These mechanisms show similar performance to the static ones. It is clear that the dynamic mechanisms that change the cache partition during the execution of a computation phase will react to phase changes and converge quicker to the optimal partition. However, there are problems with benchmarks that exhibit large IPC variations during their execution. This point may seem counter-intuitive, but optimal decisions at one point of a computation phase may not be the optimal partition for the entire computation phase. This is the case of `apsi` in pair HH2 with `parser`: In the first 54.6% of the time, `apsi` has an IPC of 0.94 instructions per cycle. In the remaining 45.4% of the time it has an average IPC of 3.59 instructions per cycle. Thus, dynamic algorithms try to assign as much cache space as possible to `apsi` because it is assumed to be the bottleneck of the parallel application. However, this is a wrong decision as later it automatically catches up `parser`, which is the real bottleneck in this pairing. When benchmarks have a similar behavior during all their execution, the dynamic mechanisms obtain better results.

On average, the performance benefits obtained with static balancing algorithms are slightly better than the dynamic mechanisms (between 2.3% and 2.7% in execution time and between 2.5% and 3% in imbalance percentage). For that reason, we conclude that using a static mechanism is more suitable to balance parallel applications.

5.4 Summary

In general, the largest reduction in execution time is obtained when a thread with more cache accesses and less cache utility is executed with a thread with more cache utility and less cache accesses. In this situation, an eviction policy like LRU cannot restrict the cache space assigned to the thread with less cache requirements, harming the performance of the other thread. Next, we list the main findings of this section.

1) When threads are not cache sensitive, adjusting the cache space devoted to each thread has nearly no impact on performance.

2) When threads have different cache behavior and the bottleneck thread is more sensitive to cache space, we can manage to improve total execution time by assigning more cache resources to the bottleneck thread.

3) When threads have similar cache behavior, performance speed ups are obtained when all threads are high utility.

4) Cache partition decisions should be made at the beginning of an iteration of the application, as the load balancing algorithms provide better results when they have a global vision of the imbalance of the application.

6. PERFORMANCE EVALUATION WITH A PARALLEL HPC APPLICATION

6.1 Extracting a Representative Trace from a Parallel HPC Application

In this section, we evaluate our balancing algorithms with real traces from a parallel HPC application running on an actual supercomputer: `wrf`. The Weather Research and Forecasting (`wrf`) model [17] is a mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. In this experiment, we use the non hydrostatic mesoscale model dynamical core.

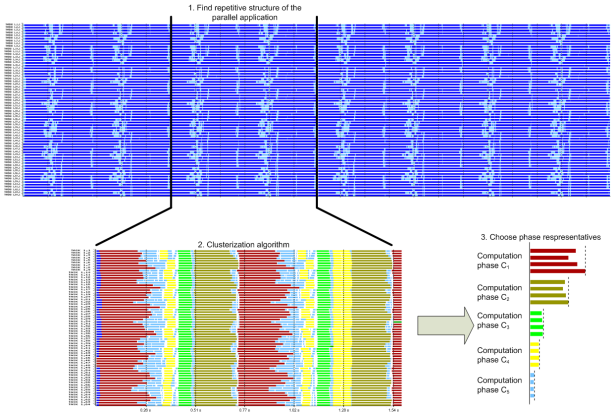


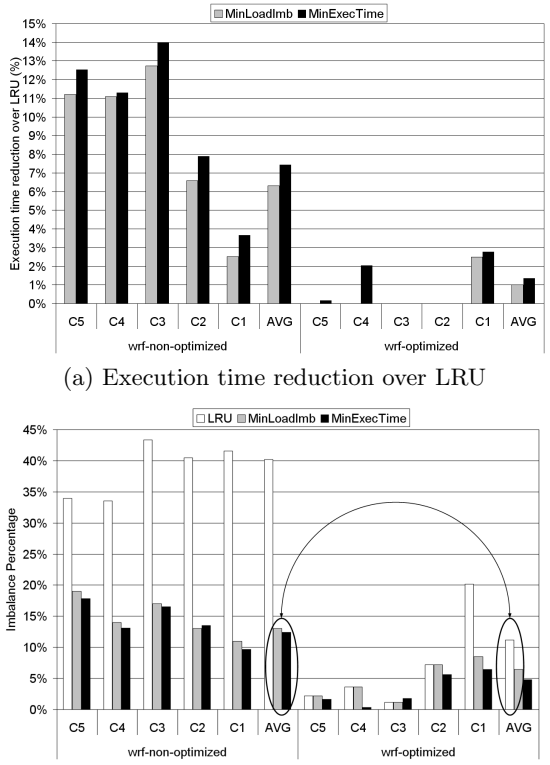
Figure 6: Experimental methodology to obtain representative traces of parallel applications. Example with `wrf` with 64 MPI processes. Four representatives are chosen per computation phase

Simulating all threads of the parallel application implies a significant amount of simulation time as these applications usually run for days or weeks on a supercomputer. We use an automatic mechanism to choose the most representative computation regions to be traced and simulated with a cycle-accurate simulator. The simulation methodology starts with a `paraver` [15] trace file generated with OMPItrace package [20]. This trace file consists of a complete timestamped sequence of events of the whole execution of a parallel application. We have used an automatic methodology to extract the internal structure of the trace [6], which allows selecting the most meaningful part of the trace file. This methodology uses non-linear filtering and spectral analysis techniques to determine the internal structure of the trace and detect periodicity of applications, which allows cutting the original parallel trace and generating a new one between 10 and 150 times shorter [6], but still in the order of minutes of real execution and not affordable for a cycle-accurate simulator.

Next, we use a clustering algorithm to determine the most representative computation bursts inside an iteration of the new trace. In [11] the authors use a density-based clustering algorithm applied to the hardware counters offered by modern processors to automatically detect the representative sections of a parallel application. This algorithm obtains R representative sections of each computation phase, where R is given by the user. As we model a CMP with four cores, we select four representatives of each computation phase with the same load imbalance as the entire phase.

We have used these reduced trace files to feed the cycle-accurate architecture simulator described in Section 4. We simulate all threads sharing the L2 cache in a CMP architecture. When a thread finishes executing, it waits until all other threads have also finished. Finally, the obtained speed ups in the computation phases execution time are passed to `dimemas` [10] (a high-level MPI application simulator) to estimate the total execution time of the parallel application.

Figure 6 exemplifies this methodology for a real application running on a supercomputer. We execute `wrf` with 64 MPI processes. On the top of the figure, we show the beginning of the original trace. First, we determine the periodicity of the application and cut the meaningful part of the trace. This process reduces an original trace from 5.06GB to 38.7MB. Next, the clustering algorithm detects five representative computations phases. For example, computation phase C_1 is executed twice during one iteration of the application. Finally, we choose 4 representative processes of each computation phase with the same load imbalance as the original phase. Note that selecting one representative for each phase reduces the simulation time by a factor of 2.5x in this application (we simulate only 40% of the total time of an iteration), while choosing 4 representatives out of 64 gives a 16x speed up.



(a) Execution time reduction over LRU

(b) Imbalance Percentage

Figure 7: Imbalance metrics when using LRU and the balancing algorithms with `wrf` (CMP architecture with 4 cores and a shared 1MB 16 ways L2 cache)

The described simulation methodology significantly reduces the time to obtain an estimation of execution time. In the case of a real application like `wrf` running with 64 threads, this methodology reduces simulation time by three orders of magnitude (5300x speed up). Using the selected traces on a single processing machine, we would need approximately half a day to estimate the speed up of a configuration, which would make it unaffordable to simulate the whole application (we would need more than 100 days of simulation time).

6.2 Case Study with a Real HPC Application

Using the methodology described in the previous section, we extracted a trace from an execution of `wrf` with 64 MPI processes and obtained 4 representatives for the 5 computation phases that compose the application.

In this evaluation, we use two versions of the same application: `wrf-non-optimized` and `wrf-optimized`. The former is a version of the application that has not been optimized for our supercomputer infrastructure. Consequently, it suffers from the load imbalance problem: the average imbalance percentage is 41.6%. After a long optimization process, requiring several man-years of effort, the application has been heavily tuned to solve this problem and reduce the imbalance percentage to 11.2%; we call this version `wrf-optimized`.

Figure 7(a) shows the reduction in execution time obtained with *MinLoadImb* and *MinExecTime* with the two versions of the parallel application. In case of `wrf-non-optimized`, the execution time with *MinExecTime* is consistently reduced, reaching 14% speed up in phase C_3 and 7.4% in average (average results take into consideration the weight of each phase in the application). The same behavior is observed with *MinLoadImb*, although the performance speed ups are not as good because the mechanism converges to the optimal partition more slowly. *MinExecTime* also improves the performance of `wrf-optimized`, in which a significant effort from the application writers was devoted to balance it, by 1.4%. The mechanism is not activated in phases C_2 and C_3 (LRU is maintained for these phases). For *MinLoadImb*, the balancing mechanism is only activated in phase C_1, obtaining an average 1.0% reduction in execution time.

Figure 7(b) shows the improvements in imbalance percentage obtained with our mechanisms. In case of `wrf-non-optimized` the imbalance percentage is consistently reduced, from an average of 41.6% to 12.4% (*MinExecTime*) and to 13.0% for *MinLoadImb*. Phase C_3 was the most unbalanced computation phase when using LRU, which explains the large speed ups in execution time that were obtained. In case of `wrf-optimized`, the original 11.2% imbalance percentage is reduced to 5.1% for *MinExecTime* and to 6.5% for *MinLoadImb*. The most important conclusion of Figure 7(b) is that our balancing algorithms reduce the imbalance of `wrf-non-optimized` to the same values of `wrf-optimized`. That is, in both cases the imbalance percentage is similar, which means that, in balancing the non-optimized code, we reach the same imbalance percentage with the optimized code, without having to spend several man-years of effort in changing the code of the application.

7. CONCLUSIONS

In this paper we have developed two load balancing algorithms for parallel applications that make use of a dynamic cache allocation mechanism to balance the application.

Our balancing algorithms have a learning phase in which the mechanism monitors the usage of the shared LLC in a CMP, determining whether it is useful to trigger cache partitioning or keep using LRU. When triggered, our algorithms assign more cache space to the slowest threads of the application. We have suggested that these assignments should be done at the end of each computation phase, as finer granularities may have problems with workloads that exhibit large IPC variations during their execution. Our proposed load balancing algorithms have been evaluated through an extensive set of experiments with synthetic workloads, reducing the execution time by up to 28.9%.

We have also applied the proposed algorithms to a real HPC application. The balancing algorithms reduce the imbalance of `wrf-non-optimized` to 12%, the same value as in `wrf-optimized` in which the optimization phase required several man-years of effort. Overall we have obtained a 7.4% execution time reduction. This is encouraging because it indicates that our dynamic balancing algorithms may represent a one-time effort that may considerably reduce the development time invested in balancing HPC applications.

8. ACKNOWLEDGMENTS

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN2007-60625 and grant AP-2005-3318, by the HiPEAC European Network of Excellence and by the SARC European Project.

9. REFERENCES

[1] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. The MPsim Simulation Tool. Technical Report UPC-DAC-RR-2009-7, january 2009.

[2] C. Boneti, F. J. Cazorla, R. Gioiosa, C.-Y. Cher, A. Buyuktosunoglu, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *ISCA*, 2008.

[3] C. Boneti, R. Gioiosa, F. J. Cazorla, J. Corbalan, J. Labarta, and M. Valero. Balancing HPC Applications Through Smart Allocation of Resources in MT Processors. In *IPDPS*, 2008.

[4] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero. A dynamic scheduler for balancing HPC applications. In *SC*, 2008.

[5] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.

[6] M. Casas, R. M. Badia, and J. Labarta. Automatic structure extraction from MPI applications tracefiles. In *Euro-Par*, 2007.

[7] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, 2007.

[8] J. Corbalan, A. Duran, and J. Labarta. Dynamic load balancing of MPI+OpenMP applications. In *ICPP*, 2004.

[9] A. Duran, M. Gonzàlez, and J. Corbalán. Automatic thread distribution for nested parallelism in openmp. In *ICS*, 2005.

[10] S. Girona and J. Labarta. Sensitivity of performance prediction of message passing programs. *J. Supercomput.*, 17(3):291–298, 2000.

[11] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic detection of parallel applications computation phases. In *IPDPS*, 2009.

[12] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.

[13] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. Steely, and J. Emer. Adaptive insertion policies for managing shared caches on CMPs. In *PACT*, 2008.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[15] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In *Euro-Par*, 1996.

[16] Metis. Family of multilevel partitioning algorithms. http://www.cs.umn.edu/metis.

[17] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The Weather Reseach and Forecast Model: Software Architecture and Performance. In *ECMWF*, 2004.

[18] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Explaining dynamic cache partitioning speed ups. *IEEE CAL*, 2007.

[19] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Online prediction of applications cache utility. In *IC-SAMOS*, 2007.

[20] OMPItrace tool. Instrumentation of combined OpenMP and MPI applications. `http://www.bsc.es/media/1382.pdf`.

[21] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[22] L. D. Rose, B. Homer, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *Euro-Par*, 2007.

[23] R. Sakellariou and J. R. Gurd. Compile-time minimisation of load imbalance in loop nests. In *ICS*, 1997.

[24] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Euro-Par*, 2000.

[25] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[26] Standard Performance Evaluation Corporation. SPEC CPU 2000 benchmark suite. `http://www.spec.org`.

[27] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.

[28] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *CASES*, 2005.