Pandia: comprehensive contention-sensitive thread placement

Daniel Goodman
Oracle Labs, Cambridge, UK
daniel.goodman@oracle.com

Georgios Varisteas *
SnT, University of Luxembourg
georgios.varisteas@uni.lu

Tim Harris
Oracle Labs, Cambridge, UK
timothy.l.harris@oracle.com

Abstract

Pandia is a system for modeling the performance of in-memory parallel workloads. It generates a description of a workload from a series of profiling runs, and combines this with a description of the machine's hardware to model the workload's performance over different thread counts and different placements of those threads.

The approach is "comprehensive" in that it accounts for contention at multiple resources such as processor functional units and memory channels. The points of contention for a workload can shift between resources as the degree of parallelism and thread placement changes. Pandia accounts for these changes and provides a close correspondence between predicted performance and actual performance. Testing a set of 22 benchmarks on 2 socket Intel machines fitted with chips ranging from Sandy Bridge to Haswell we see median differences of 1.05% to 0% between the fastest predicted placement and the fastest measured placement, and median errors of 8% to 4% across all placements.

Pandia can be used to optimize the performance of a given workload—for instance, identifying whether or not multiple processor sockets should be used, and whether or not the workload benefits from using multiple threads per core. In addition, Pandia can be used to identify opportunities for reducing resource consumption where additional resources are not matched by additional performance—for instance, limiting a workload to a small number of cores when its scaling is poor.

1. Introduction

Pandia is a system for modeling the performance and resource demands of parallel in-memory workloads. From a series of six carefully-selected profiling runs, Pandia quantitatively predicts a workload's performance across different numbers of threads and different placements of those threads within a machine. Pandia's results can be used both to predict the best thread allocation for a given workload, and to predict the resources needed for a workload to meet a specified performance target.

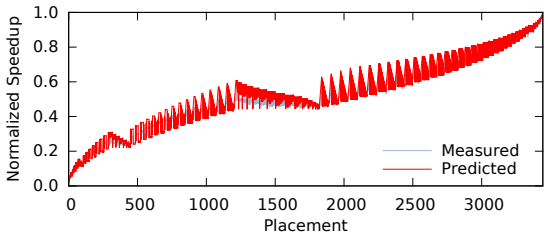


Figure 1: Measured performance vs predicted performance for MD, a molecular dynamics simulation. The x-axis shows the different thread placements explored, from 1 thread up to the full 2-socket machine with 72 hardware threads. For each placement we plot the execution time normalized to the best performance achieved.

Figure 1 shows an example set of results from Pandia collected on a molecular dynamics simulation (MD) running on a 2-socket Intel Haswell system. The x-axis shows different thread allocations, exploring different numbers of threads and different placements of those threads on the hardware contexts in the machine. The lighter (gray) points show the measured performance of the workload when testing all of these allocations. The darker (red) points show the performance predicted by Pandia.

Pandia exploits characteristics which are typical of in-memory parallel analytics workloads: there is a fixed amount of computation which can be distributed across a configurable number of threads. By varying the placement of these threads, we can control how the workload's fixed demand for resources maps onto the underlying machine. Unlike more general server workloads, there is little OS activity, and there are simple synchronization patterns across the threads.

Pandia benefits from current trends in systems and processors. For instance, the use of fully-connected symmetric interconnects removes some of the complexities seen in prior work [18]. In addition, the introduction of features such

* Work conducted while an intern at Oracle Labs and a PhD candidate at KTH Royal Institute of Technology.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064177

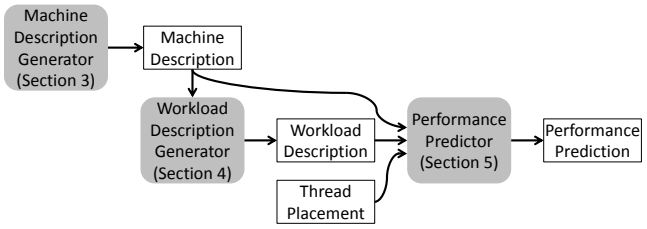


Figure 2: The three main components of Pandia and their dependencies.

as adaptive caches provide more gradual fall-offs in performance if an application's working set grows to exceed a given level in the cache hierarchy [27].

In addition to discussing these assumptions and trends in more detail, Section 2 provides an overview of the components that make up Pandia: We generate a machine description (Section 3), generate a description of a workload via a series of profiling runs (Section 4), and from these we model the performance of a given thread placement (Section 5). Figure 2 illustrates these steps.

In outline, we start by running a workload with a single thread and measuring its use of different resources (e.g., memory bandwidth). Additional runs explore the workload's behavior when running with larger numbers of threads, and its sensitivity to different changes in thread placement—for instance, the impact of co-locating threads onto the same core, or the impact of using cores on the same socket versus different sockets. We need six runs in total.

Based on these measurements, Pandia then makes predictions by naïvely assuming that each thread in the parallel workload will impose the same load on the machine as the single thread in the initial profiling run. We then iteratively refine this load by (*i*) identifying the bottleneck resources for each thread, and scaling back the thread's performance so that the bottleneck is not oversubscribed, and (*ii*) predicting the overheads incurred by communication and synchronization, and scaling back the thread's performance further to account for these costs. There can be complex interactions between the demands for different resources because slowing some threads may enable threads with different bottleneck resources to run faster. We therefore iterate until a stable result is achieved.

Section 6 evaluates the performance of Pandia on 22 test workloads. We split these workloads into a reference set of 4 which we studied in detail while developing Pandia, and a test set which we added to assess the generality of the techniques developed. While we evaluate Pandia using standalone benchmarks our ultimate aim is to support parallel workloads within a server applications, for instance, traditional SQL data analytics, or parallel graph analysis.

Section 7 discusses related work before Section 8 concludes this paper.

Contributions of this work. The goal of this work is to predict the performance of parallel workloads on shared-memory multi-core/multi-socket machines. We make the following contributions:

- We identify a set of hardware and software assumptions which permit performance predictions to be made without requiring detailed micro-architectural models for the cache or other resources. In contrast, prior work has generated miss-rate curves (e.g., that of Chandra *et al.* [8]), or needed to place workloads into a taxonomy based on their interactions in shared caches (e.g., that of Lin *et al.* [19], or of Xie and Loh [30]).
- We develop techniques for predicting performance based on iteratively predicting the slowdowns that a workload will experience due to contention-based overheads and synchronization-based overheads.

2. Overview and assumptions

This section takes an initial look at each of the components that makes up Pandia (Section 2.1), the assumptions we make about the target hardware (Section 2.2), and the software workloads we consider (Section 2.3).

2.1 Components of Pandia

Pandia comprises three components (Figure 2). We summarize them briefly:

Machine description generator (Section 3). This is a collection of "stress applications" designed to saturate different resources in the machine. We measure the performance of these applications with hardware counters. From these measurements we can determine the structure of the machine, and properties such as the maximum bandwidth at different levels of the memory hierarchy. The results form a machine description for use by subsequent modeling steps.

Workload description generator (Section 4). Pandia executes a workload in six carefully-selected profiling runs. We record results including CPU performance counters for (*i*) the workload's solo performance, (*ii*) the workload's scaling across resources, and (*iii*) when co-located with stress applications, the interference these stressors impose on the workload.

Performance predictor (Section 5). Finally, Pandia takes a machine description, a workload description, and a proposed thread placement. From these it provides a prediction of the workload's resource demands and performance.

2.2 Target hardware environment

Pandia is designed for cache-coherent shared-memory multiprocessor machines. Our evaluation uses 2-socket and 4-socket Intel Xeon systems with multiple threads within each core, multiple cores within each chip, and multiple chips within the complete machine. We assume that the hardware is homogeneous in that each core is identical, each chip is identical, and the interconnect is fully connected.

Current server hardware has removed many of the discontinuities in behavior that occurred in earlier systems. For instance, recent Intel systems up to 4 sockets and SPARC systems up to 8 sockets normally have fully-connected interconnects with identical amounts of memory on each socket. Modern systems have caches which monitor cache usage and modify the insertion and removal strategies accordingly [27]. This removes many pathological effects when the working set outgrows the cache.

The overall effect of these changes is that we are able to construct a performance model using simpler techniques than were required to model previous generations of hardware. In fact, without complex hardware insight, many techniques used on previous generations of hardware no longer appear to work (for instance, measuring cache sizes using microbenchmarks to seek "performance cliffs").

2.3 Software workload and models

We make the following assumptions about the workload:

- A generally-parallel workload with a configurable number of threads, and plentiful work to share between them.
- Homogeneous behavior between the threads—e.g., if executing loop iterations in parallel, there are similar resource demands for each iteration.
- Low algorithmic cost of adding extra threads. For instance, expanding the degree of parallelism should not significantly extend sequential sections of the code.
- Little system call activity or I/O. This simplifying assumption reduces the kinds of behavior we need to model, and it appears valid in our in-memory analytics workloads. We aim to explore broadening our scope in future work.

These properties cover many analytics workloads where there is a constant amount of "work" to perform (e.g., a given graph's vertices to iterate over, or a given database column to process). In our evaluation we use a range of in-memory database join operators, a graph analytics workload, and existing parallel computing benchmarks written using OpenMP [26].

In some of our workloads threads do perform busy-waiting while delayed at synchronization barriers or for brief critical sections. Empirically, we are able to ignore the impact of the additional instructions spent spinning: good implementations consume few CPU pipeline resources while waiting, and waiting terminates quickly when the waited-for resource becomes available.

Given such a workload, we model its behavior using six metrics. In summary:

Resource demands. We model the hardware resources that we expect threads to consume. These resources include bandwidths at each level of the memory hierarchy, and the compute resources within a core.

Parallel fraction. We model the fraction of a workload which runs in parallel (p). From this we can derive a basic speedup value via Amdahl's law [1] when executing with n threads:

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$

Given our assumption of homogeneous behavior across threads, we assume that the non-parallel work is spread evenly over the workload—for instance, work executing in critical sections rather than one long sequential portion in a "main" thread.

This is of course a crude approximation of the workload's actual scaling because it assumes no interference between the parallel sections of the code—other aspects of Pandia seek to account for these more complex effects.

Inter-socket communication. The bandwidth consumed on inter-socket links is recorded as part of the workload's resource demands. In addition, we explicitly model the impact of the longer latency introduced by crossing sockets.

When threads on different sockets access common memory locations the performance depends on many factors such as the relative locations of the threads, the kinds of access being made, and the presence or absence of false sharing.

The approach we take in Pandia is to measure these effects in aggregate by determining the workload's sensitivity to having its threads placed on different sockets. This captures the sensitivity of the complete workload to these costs. For instance, if threads communicate rarely then the costs may remain low even if the hardware introduces high latencies on each communication between sockets.

Load balancing. Although we assume a workload's threads have homogeneous behavior, some of these threads may run more slowly than others (e.g., some threads may share cores, while others do not). In these cases we must assess how performance skews between threads affect the overall workload.

For instance, some workloads use static work distribution, and a slow thread becomes a straggler and delays the whole workload. In this case each of the threads performs an equal amount of work, but the time that the threads spend performing useful work may differ.

Other workloads use dynamic load balancing to redistribute work, allowing any slowness by one thread to be compensated for by other threads picking up its work. In this case the overall performance of the workload is determined by the aggregate throughput of all of its threads. Threads will be active for the same amount of time, but some threads will be making progress more quickly than others.

We quantify this by a *load balancing factor* indicating where a workload lies between these extremes. In practice, workloads may be somewhere between these points.

Core burstiness. Core burstiness quantifies the extent to which the workload's demands for resources in a core are

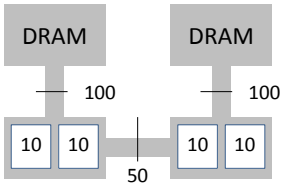


Figure 3: Machine description for a simple system with two dual-core processors and no caches. The model indicates the bandwidth on the memory links and the interconnect, along with the maximum instruction throughput per core.

spread evenly over time or occur in bursts. Bursts can produce additional resource contention when threads are collocated. It can be misleading to rely on simple average demands captured over a time: a low average may be a steady demand which can be accommodated from multiple threads without penalty, or it may be peaks and troughs.

Thread utilization. If applications fail to scale perfectly then each thread is busy for less than 100% of its time. For example, reduced utilization may be due to sequential sections of the workload, or to delays waiting on stragglers. We denote this utilization factor f and Pandia uses it to adjust the resource demands of each thread: if a thread is busy 50% of the time then it will demand 50% less resources than if it is busy 100% of the time.

Unlike other aspects of our models, values of f differ between threads and depend on the workload, the machine description, and the thread placement being explored. As we illustrate in Section 5, we derive f dynamically from these other sources when making performance predictions.

3. Machine description

This section describes how Pandia collects the information required to model the structure and performance of a machine. The resulting machine descriptions are workload-independent and are created once for each machine.

The machine description is constructed through a combination of information obtained from the operating system (OS), and measurements taken from synthetic applications which stress particular resources. The OS provides:

- The number of processor sockets, cores per socket, and hardware threads per core.
- The structure of the links between each level in the cache hierarchy, the topology of the interconnect between the sockets, and the locations of the links to main memory.

This initial topology is represented as a graph of the hardware components of the machine and the relationships between them. We combine this with performance measurements describing the bandwidth of the different links in the memory system and the performance of the cores.

Figure 3 shows an example machine description for a simple system without caches. Note that for brevity we omit units: so long as consistent units are used when modeling the machine and the workload, the exact scale is not significant.

3.1 Measuring link bandwidth

Starting from the machine topology, we run stress applications to determine the maximum bandwidth achieved on each link in the memory system (e.g., between each level in the cache hierarchy, or on the interconnect between sockets). For all of these measurements we use results obtained from workloads running on the machine itself rather than numbers obtained from data sheets. This empirical approach lets us use the same measurement techniques for profiling the machine and for profiling the performance of a workload.

The stress applications allocate an array of parameterizable size which is then accessed linearly with one value read and/or written per cache line. The accesses are in an optimized loop with constant arguments to allow for effective prefetching and branch prediction. When multiple threads are used, each thread has a unique set of cache lines that it will access. The size of the array is chosen to almost fill the size of the storage at the far end of the link whose performance we are measuring (without growing so large as to spill into the next level). In the case of main memory we make the array at least 100 times the size of the last level of cache (LLC). This ensures that almost all of the accesses miss in the cache LLC. When placing data into memory tools such as Linux `numactl` are used to control placement.

When measuring the bandwidth of shared caches, it is important to measure the maximum bandwidth of each link to the cache and the maximum cumulative bandwidth that the cache can sustain. For example a L3 cache may not be able to support sufficient bandwidth for the maximum access rate from a core to be sustained to all cores simultaneously. So on an 18-core chip each core may individually achieve a peak bandwidth of 360, but the L3 cache as a whole may only provide 5000, not 6480. Both limits form part of the machine description: 360 per core, and 5000 in aggregate.

3.2 Measuring core performance

The maximum instruction rate of a core is measured from performance counters while executing a CPU stress application. This performs operations on a small dataset such that it readily fits into the L1 cache. Pipeline stalls are avoided by providing a large number of independent operations, and branch stalls are reduced by unwinding the loop and using constant values on the loop guard to allow good prediction.

The operations are integer based to enable peak performance. Performance is measured in instructions executed per unit time. Although there is variation in peak performance based on the type of operation used in the loop, this measurement appears to be sufficient for our workloads.

We also execute two threads on the core to assess if the core suffers a loss in peak performance when co-scheduling threads (e.g., due to contention in the front-end).

Step	Property	Example
1	**Single thread resource demands** (d) – The single-thread execution time t_1 and a vector of resource demands for that one thread.	$[7, 40]$
2	**Parallel fraction** (p) – The fraction of the workload which runs in parallel.	0.9
3	**Inter-socket overhead** (o_s) – The latency relative to t_1 for inter-socket communication when threads are placed on different sockets.	0.1
4	**Load balancing factor** (l) – The extent to which the workload can re-balanced dynamically between threads based on their progress.	0.5
5	**Core burstiness** (b) – The sensitivity to collocation of threads in a core.	0.5

Figure 4: Steps in creating a workload model, along with possible values for an example on the machine in Figure 3.

4. Workload model

We now describe how Pandia collects the information required to model a given workload. The resulting workload description is specific to a given machine and, ideally, it will be regenerated when moving to different hardware. However, as we show in our evaluation, predictions can remain useful across broadly similar hardware platforms (e.g., a workload that scales poorly across sockets on one system may well scale poorly across sockets on another).

We build up the workload model incrementally in 5 steps, with each step providing a successively more detailed description of the workload's behavior. We organize the experimental runs so that the behavior being considered in a given step depends only on factors already determined in preceding steps. Step 2 is dependent on step 1, and steps 3-5 are dependent both on step 1 and on step 2. Aside from step 1 these dependencies exist only in the calculation of the model parameters; the actual experimental runs can proceed concurrently if multiple machines are available. Figure 4 shows the steps, the properties collected, and example values.

Thread utilization. As we described in Section 2.3, if a thread suffers contention its resource requirements will decrease. This is modeled by thread utilization. Thread utilization does not form an explicit part of the workload description because it is dependent on the thread placement being explored, but is introduced here because it is necessary to account for its effect when choosing thread placements. In step 1 when running with a single thread by definition there will be no thread contention. In step 2 we choose the thread placement to avoid thread contention, as such there will also be no contention in step 4. In steps 3 and 5 however, contention is likely. We briefly describe how thread utilization is defined and derived when required, before describing the experiments carried out for the 5 workload modeling steps in turn (Section 4.1–4.5).

Under our workload assumptions, if a workload does not scale perfectly, then its execution time will increase while the total useful instructions it executes will remain broadly

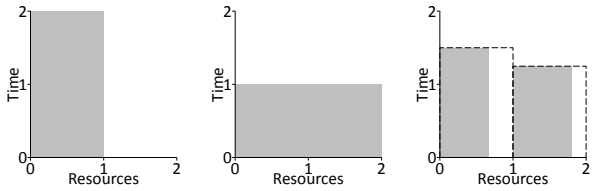


Figure 5: Examples of possible executions of threads running a given workload. The gray boxes show the resources used by the workload, spread over different numbers of threads (x-axis) and durations (y-axis).

constant. This means the rate of resource consumption for threads will be reduced because the operations performed by the threads are being stretched over longer time intervals. Figure 5 illustrates this. The first graph shows an original single threaded run executing for 2 units of time. The second graph shows ideal scaling with two threads executing for 1 unit of time: the per-thread consumption of resources is at the same rate as the original. The third graph shows a more realistic example with the two threads being held up by contention: the resource demands are stretched out over time. The thread utilization factor quantifies this as the ratio between the resources used by the threads (grey boxes) and the resources available during this interval (dashed outlines).

In our calculations we denote the thread utilization factors f. Each thread has its own value, which we re-calculate whenever the predicted performance of the thread changes. We write f_x for the thread utilization factors calculated at the start of step x.

4.1 Single thread time and resource demands (Step 1)

First, the workload is run with a single thread to get the sequential time t_1 along with a vector of resource demands comprising the instruction execution rate and the bandwidth requirements to each level of the cache hierarchy and to main memory. These metrics provide the basic sequential resource demands of the workload. They are measured using the same performance counters that we used in Sections 3.1 and 3.2.

Figure 6 (Run 1) shows an example of the results that we collect in the simplified example machine: an instruction rate of 7, and memory transfer bandwidth of 40 to each socket.

We use the single thread performance and resource demands as a reference point in subsequent steps. We normalize execution times for subsequent experiments t_x relative to this sequential execution time such that $r_x = t_x/t_1$.

In each of the subsequent runs, we consider these relative times r_x to be the product of two values: the *known factors* (k_x) already accounted for by previous steps, and the *unknown factors* (u_x) which are yet to be determined. The incremental approach we take is to layer steps so that, for a given step, we can use Pandia based on the partial workload model already constructed to determine k_x. We then extend the workload model so that $u_x = r_x/k_x$ is predicted

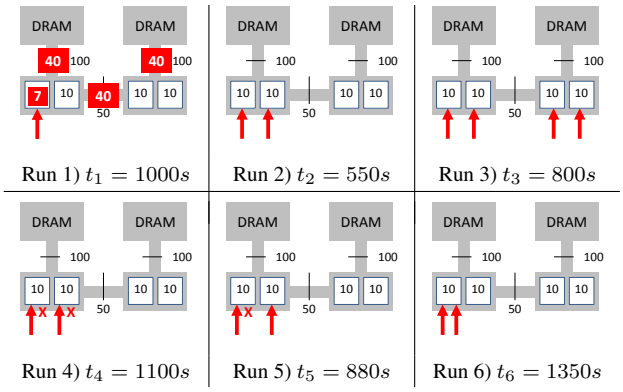


Figure 6: The six runs used to generate a description of the example workload. Arrows represent threads and crosses represent stress applications added to perturb execution.

correctly with the inclusion of the results of the new step. Thread utilization factors f are recomputed after each step.

4.2 Parallel fraction (Step 2)

The parallel fraction is determined with an extra run (Run 2 in Figure 6). We construct the thread placement carefully to avoid contention: (*i*) we run one thread per core, (*ii*) we keep the threads within a single socket (to avoid considering inter-socket communication costs at this point), and (*iii*) based on the resource information from Run 1, we set the number of threads sufficiently low to avoid over-subscribing any resources (so thread utilization factors $f = 1$). We use the largest even number of threads we can while satisfying these conditions (as we show later, using an even number of threads lets us re-use this run in subsequent steps). In practice, we can span most of the cores in one socket on the machines we use.

This careful thread placement means that there is no resource contention: the known factors from the existing model are $k_2 = 1$, and we have $u_2 = r_2$. Given u_2 from the timing of the run, we calculate the parallel fraction by finding p using Amdahl's law such that:

$$u_2 = 1 - p + \frac{p}{n}$$

4.3 Inter-socket latency (Step 3)

Following our workload assumptions, we take each thread to communicate equally with every other thread. In our models, we define the inter-socket overhead o_s as the additional time penalty a thread incurs for each of the threads on a different socket to itself. To derive this we take an even number of threads and split them across the two sockets (Run 3 in Figure 6). This ensures that each thread sees the same number of cross-socket links, avoiding the need to consider results from subsequent steps in the model.

From the partial workload model of steps 1–2 we calculate k_3 (the speedup expected by Amdahl's law and the slowdown from resource contention), and we calculate the thread utilization f_3 based on the execution time predicted by these steps, k_3. Given the $n/2$ inter-socket links per thread, we get:

$$r_3 = u_3 \times k_3 = \left(1 + \frac{\frac{n}{2} \times o_s}{f_3}\right) \times k_3$$

Hence given the existing workload model and r_3 from the timing of the run we can solve for o_s in:

$$u_3 = 1 + \frac{\frac{n}{2} \times o_s}{f_3}$$

4.4 Load balancing factor (Step 4)

The profiling runs for steps 1–3 use symmetric thread placements so all threads proceed at the same rate. We now extend the workload description to describe cases where threads are not placed symmetrically. In these cases, it is important to determine the effect on the overall speed of a workload if some threads slow down more than others. For instance, some workloads use static work distribution, and all threads must wait until the slowest one completes. Other workloads use work stealing to distribute work dynamically between threads, allowing any slowness by one thread to be compensated for by other threads picking up its work.

We express this by a *load balancing factor* $l \in [0 \ldots 1]$ indicating where a workload lies between these extremes. If $l = 0$ then there is no dynamic load balancing, and the threads proceed in lock-step. If $l = 1$ then they dynamically redistribute work. In practice, workloads may be somewhere between these points.

We determine l by running experiments to show how the performance of one thread impacts the performance of the complete workload. To do this we observe how the workload's performance changes when we co-schedule stress applications alongside some of its threads. In a given run, we write s_i as the slowdown added to thread i. If there are n threads, and a parallel fraction p, then the relative execution rate in the two extreme cases is:

$$\text{Lock-step:} \quad s_{\text{lock}} = \left((1-p) + p \times \max_{i=1}^{i=n} s_i\right)$$

$$\text{Load-balanced:} \quad s_{\text{bal}} = \left((1-p) + np \Big/ \sum_{i=1}^{n} \frac{1}{s_i}\right)$$

For a run between these extremes we have:

$$s_l = (1-l) \times s_{\text{lock}} + l \times s_{\text{bal}}$$

We calculate l from Runs 2, 4, and 5 in Figure 6. In Run 2 the threads execute as normal, so for all i, $s_i = 1$. In Run 4 all threads compete against a simple CPU-bound loop which will delay their execution. We determine the slowdown introduced by this delay as u_4/u_2. This indicates the penalty of slowing down all threads without causing load imbalance. Using this measured slowdown we calculate the extreme points s_{lock} and s_{bal} for the case where $n-1$ threads have $s_i = 1$ and 1 thread has $s_i = u_4/u_2$.

Finally, in Run 5, we measure the performance of the workload in practice with one thread slowed. This provides $s_l = u_5/u_2$, letting us solve for l by interpolating between s_{lock} and s_{bal}.

4.5 Core burstiness (Step 5)

Finally, to account for bursty resource demands, we compare the performance of two runs which differ only in the collocation of threads on cores (Runs 2 and 6 in Figure 6). Run 2 uses one thread per core across a single socket, while Run 6 uses the same number of threads packed into half the cores.

We calculate the thread utilization factor f_6 for run 6 based on the value k_6 which Pandia generates from the partial workload model from steps 1–4. Because of Run 2's constraints $f_2 = 1$. We then take the remaining unknown factors in these two runs and define burstiness as the percentage extra time required due to collocation:

$$\text{Burstiness:} \quad b = \frac{1}{f_6} \times \left(\frac{u_6}{u_2} - 1 \right)$$

5. Performance prediction

Given a machine description and a workload description we now show how Pandia predicts the performance for a proposed thread placement. We demonstrate this using the worked example in Figure 7 placing the workload profiled in Section 4. The performance is constructed from two elements: (*i*) an anticipated speed-up based on Amdahl's law assuming perfect scaling of the parallel section of the workload, and (*ii*) a slowdown reflecting the impact of resource contention, communication, and synchronization.

Speedup. As discussed earlier, the speedup via Amdahl's law is calculated in the usual way based on the parallel fraction of the workload (p) and the number of threads in use (n). For the workload in Section 4, $p = 0.9$, and in the example in Figure 7, $n = 3$, giving a speedup of 2.5.

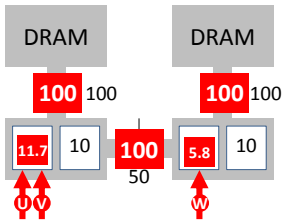
Slowdown. The slowdown is then predicted by considering the resource-contention, communication, and synchronization introduced by the threads. These factors are interdependent: if a thread is slowed by synchronization then this will reduce the pressure it places on the memory system. That reduction may in turn enable another thread to run more quickly, which may change the synchronization behavior of the workload, and so on. We handle these different factors by proceeding iteratively until a stable prediction is reached (in practice only a few iteration steps are needed for the workloads we have studied).

Figure 8 illustrates the overall approach: Pandia calculates a predicted slowdown for each thread from resource contention, along with predicted penalties from cross-socket communication and from poor load balancing. In addition, it maintains a thread utilization factor to scale the resource demands according to the fraction of the time the thread is working. Figure 7(a) illustrates this for a running example with three threads running the workload from Section 4.

We initialize the thread utilization factors as the Amdahl's law speedup divided by the number of threads. This reflects the fraction of the time that a thread would be busy if the Amdahl's law speedup is achieved—in our example, if $n = 3$, and the Amdahl's law speedup is 2.5, then the threads will

Thread	U	V	W
Resource slowdown	1.00	1.00	1.00
+ communication penalty	0.00	0.00	0.00
+ load balance penalty	0.00	0.00	0.00
Overall slowdown	1.00	1.00	1.00
New thread utilization	0.83	0.83	0.83

(a) State at the start of the first iteration.



(b) Naïve resource demands for three threads U, V, W at the start of the first iteration.

Thread	U	V	W
Resource slowdown	**2.83**	**2.83**	**2.00**
+ communication penalty	0.00	0.00	0.00
+ load balance penalty	0.00	0.00	0.00
Overall slowdown	2.83	2.83	2.00
New thread utilization	0.29	0.29	0.42

(c) Slowdowns updated based on the most over-subscribed resource used by each thread, and also to reflect the fact that U and V share a core.

Thread	U	V	W
Resource slowdown	2.83	2.83	2.00
+ communication penalty	**0.03**	**0.03**	**0.08**
+ load balance penalty	0.00	0.00	0.00
Overall slowdown	2.87	2.87	2.08
New thread utilization	0.29	0.29	0.40

(d) Slowdowns updated to include predicted cross-socket communication. U and V will communicate with lower overhead in this case than U and W.

Thread	U	V	W
Resource slowdown	2.83	2.83	2.00
+ communication penalty	0.03	0.03	0.08
+ load balance penalty	**0.00**	**0.00**	**0.40**
Overall slowdown	2.87	2.87	2.48
Utilization	0.29	0.29	0.34

(e) After the first iteration, slowdowns updated to include the effect of dynamic load balancing between threads.

Figure 7: Example steps for three threads (U, V, W) running the workload from Section 4.

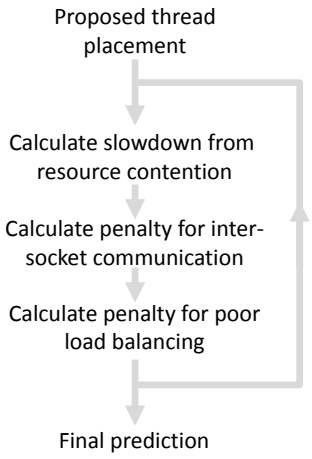


Figure 8: Overall technique, iteratively estimating the slowdown from resource contention, and the penalties incurred by inter-socket communication and poor load balancing.

be busy in parallel work for 83% of their time. We call this first estimate f_{initial}. Note that we use the same value across all threads rather than distinguishing a "main" thread which executes sequential sections: this reflects our assumption of a generally-parallel workloads in which sequential work is scattered across all of the threads in critical sections.

We describe the three steps within each iteration in Sections 5.1–5.3, then we describe how execution proceeds from one iteration to the next (Section 5.4), and the calculation of the final prediction (Section 5.5).

5.1 Slowdown from resource contention

To model contention for hardware resources, we start from a naïve set of resource demands based on the vector d in the workload description (Figure 4). The values in the vector represent rates, and so we add them at each of the locations running a thread from the workload. We scale the values by the respective thread utilization factors. So, for example, while the aggregate required bandwidth to DRAM is $3 \times 40 = 120$, it is scaled so $0.83 \times 120 = 100$ (Figure 7b).

Based on the resource demands, Pandia computes the initial predicted slowdown for each thread. The slowdown is the maximum factor by which any resource used by the thread is over-subscribed. In the example, this is the interconnect link between the two sockets which is oversubscribed by a factor of $\frac{100}{50} = 2.00$. In complex examples different threads may see different bottlenecks.

In addition, we incorporate the workload model's core burstiness factor (b) whenever threads share a core. As we described in Section 2.3, this reflects the fact that some workloads show significant interference between threads on the same core even though the average resource demands for functional units are well within the limits supported by the hardware. In the example, threads U and V are slowed because they share a core, whereas W does not. As with the basic resource contention, the impact of b is scaled by the current thread utilization factors—hence U and V incur an additional slowdown of $2.00 * 0.5 * 0.83 = 0.83$, for a total 2.83 (Figure 7c). We recompute the thread utilization factors reflecting these new slowdowns.

5.2 Penalties for off-socket communication

We now account for the impact of communication between sockets by calculating *communication slowdown*, the slowdown due to communication relative to unrestricted running. Quantitatively, the overhead value o_s represents the additional latency seen by a thread for each other thread placed on a different socket. As with the other times we use, the value of o_s is relative to the single-thread execution time t_1. To predict the performance impact of communication we consider (*i*) the locations of the threads being modeled, and hence the number of pairs which span sockets, and (*ii*) the amount of work that will be performed by each thread, and hence how significant or not a given link will be (note that the profiling runs used to derive o_s were constructed so that all threads do identical amounts of work).

We define $o_{i,j}$ to be the latency incurred by thread i for communication with thread j—this is equal to o_s if the threads are on different sockets and 0 otherwise.

To model the amount of work performed by each thread we must consider the load balancing factor: if the threads proceed in lockstep then the amount of work they perform will be equal, whereas if they use dynamic load balancing then faster threads will perform more of the work. We consider the communication in these two extreme cases and then interpolate linearly between them based on the load balancing factor l:

Completely lock-step execution. When execution proceeds without load balancing, each of the threads performs an equal amount of work so the cost for thread i is:

$$\text{lockstep}(i) = \sum_{j=1}^{j=n} o_{i,j}$$

In the example:

$$\text{lockstep}(U) = \text{lockstep}(V) = 0.0 + 0.0 + 0.1 = 0.1$$

$$\text{lockstep}(W) = 0.1 + 0.1 + 0.0 = 0.2$$

Completely independent execution. When execution is completely independent, the amount of work performed by the threads may differ. The busier threads will communicate more, and their links with other threads are more significant. Given the current predicted slowdowns for each thread $s_1 \ldots s_n$, we define the weight w_i of a thread as the fraction of the total work that thread i will perform:

$$\text{work}_i = \frac{1}{s_i} \qquad w_i = \frac{\text{work}_i}{\sum_{j=1}^{j=n} \text{work}_j}$$

In the example, given slowdowns of 2.83, 2.83, and 2.00 for the three threads, we have weights of 0.29, 0.29, and

0.41 respectively. The fastest thread will perform more of the work than the slower threads, and the communication it performs is likely to be more significant.

Given these weights the communication cost is then:

$$\text{independent}(i) = n \sum_{j=1}^{j=n} w_j o_{i,j}$$

In the example:

$$\begin{aligned}\text{independent}(U) &= \text{independent}(V)\\ &= 3\times(0.29\times0.0 + 0.29\times0.0 + 0.41\times0.1)\\ &= 0.124\\ \text{independent}(W) &= 3\times(0.29\times0.1 + 0.29\times0.1 + 0.41\times0.0)\\ &= 0.176\end{aligned}$$

Combining the results. Given the extremes, we interpolate linearly between them based on the load balancing factor l:

$$\text{comm. slowdown}(i) = l\,\text{independent}(i) + (1-l)\,\text{lockstep}(i)$$

In the example with $l = 0.5$:

$$\begin{aligned}\text{comm. slowdown}(U) &= \text{comm.slowdown}(V)\\ &= 0.5 \times 0.1 + 0.5 \times 0.124\\ &= 0.112\\ \text{comm. slowdown}(W) &= 0.5 \times 0.2 + 0.5 \times 0.176\\ &= 0.188\end{aligned}$$

Each of these is then scaled by the respective thread utilization factors (0.29, 0.29, 0.42), and then added to the penalties considered so far. This leads from Figure 7c to Figure 7d, with the overall slowdowns now (2.87, 2.87, 2.08).

5.3 Penalties for poor load balancing

In the last step of each iteration, Pandia accounts for whether the workload can dynamically rebalance work between the threads (Section 4.4). In the extreme case, if work is distributed statically between threads, then they must wait for one another to complete work; the overall performance is governed by the slowest thread. In the example, thread W would be slowed down to match U and V if they operated completely in lock-step, and all three threads would have slowdown 2.87.

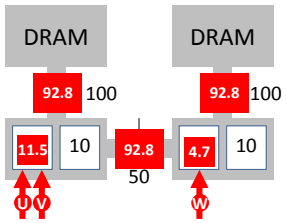
We use the workload's load balancing factor l to interpolate between this extreme case and the workload's current predicted slowdown. In the example, where $l = 0.5$ this leads from Figure 7d to Figure 7e with W being slowed down to 2.48 (i.e., the point 50% between 2.08 and 2.87).

5.4 Iterating

We need to execute multiple rounds of prediction because the penalties incurred due to communication or poor load balancing may reduce the thread's load on the system (allowing other threads to run more quickly, adding load elsewhere). In practice only a few iterations are needed for the thread utilization factors to converge in the workloads we have studied. All the values we calculate are bounded between no slowdown and the maximal slowdown experienced on the first iteration. To prevent oscillation a dampening function engages after a 100 iterations. So far we have not come across a workload that terminates as a result of this.

Thread	U	V	W
Resource slowdown	1.00	1.00	1.00
+ communication penalty	0.00	0.00	0.00
+ load balance penalty	0.00	0.00	0.00
Overall slowdown	1.00	1.00	1.00
New thread utilization	**0.82**	**0.82**	**0.67**

(a) State at the start of the second iteration.



(b) Naïve resource demands computed at the start of the second iteration.

Figure 9: Initial state for the second iteration.

Information is fed from iteration i to $i+1$ by updating the thread utilization factors used at the start of $i+1$: For each thread we determine how much of the overall slowdown in iteration i was due to the penalties incurred. This is the ratio of the thread's slowdown due to resource contention to its overall slowdown—in our example, threads U and V have $2.83/2.87 = 0.99$, and thread W has $2.00/2.48 = 0.81$. This difference reflects the fact that thread W is harmed by poor load balancing. We start the new iteration $i+1$ by resetting the thread utilization factors to f_{initial} scaled by the penalties: in effect this transfers the lessons learned about synchronization behavior in iterations $1 \ldots i$ into the starting point for iteration $i+1$.

Figure 9 shows this for our example. The thread utilizations for threads U and V are updated to $f_{\text{initial}} \times 0.99 = 0.83 \times 0.99 = 0.82$, and W is updated to $0.83 \times 0.81 = 0.67$ (Figure 9a). We reset the other parts of the prediction, and then continue by computing the new resource demands based on the new thread utilization factors (Figure 9b). Comparing the resource demands with Figure 7b, the load imposed by thread W is reduced significantly, but the interconnect remains the bottleneck.

5.5 Final predictions

Once the per-thread predictions have converged, we calculate the final predicted performance by combining the speedup from Amdahl's law with the predicted slowdowns:

$$\text{speedup} = \text{Amdahl's law speedup} \times \frac{\sum_{i=1}^{n} \frac{1}{s_i}}{n}$$

In the example this gives a predicted speedup of 1.005 after 4 iterations. This extremely poor performance is primarily due to the inter-socket link being almost completely saturated by a single thread.

6. Evaluation

We prototyped Pandia using a Python test harness to run benchmark workloads, and a Java library to perform the performance predictions. We use unmodified benchmark binaries running as stand-alone applications (although, longer-term, we envisage Pandia being used within a multi-user server application to control the management of different parallel activities within the server). Thread placement is controlled explicitly via pinning. We use Oracle Linux 6.5 with kernel version 2.6.32.

We evaluate Pandia with a set of 22 workloads. We studied 4 of these during the development of our work; the remaining 18 were used purely for evaluation. The workloads come from the NAS parallel benchmark suite (NPB) [2], the SPEC OpenMP workloads (OMP) [24], plus in-memory graph analytics workloads from our previous work [14], and database join operators from Balkesen *et al.* [3].

Some benchmarks were excluded from the NPB and OMP suites due to their execution times being either insufficient or excessive. We also exclude equake as it has a computationally intensive reduction step that significantly increases the work required every time a thread is added. This invalidates our workload assumptions from Section 2.2. We return to equake later in this section.

6.1 Two-socket machines

Our principal evaluation was carried out using a selection of two-socket Intel-based Oracle machines. The largest are Haswell systems with 18 cores per socket and 72 hardware threads in total (model X5-2). The others are Ivy Bridge systems (X4-2), and Sandy Bridge systems (X3-2), all with 8 cores per socket and 32 hardware threads in total.

To evaluate the accuracy of the predictions we ran a large number of timed runs. For the X5-2 we made 72 448 timed runs covering approximately 20% of each workload's possible placements. For each of the smaller machines we exhaustively tested the possible placements with 41 868 runs. This took 153, 82, and 107 machine-days for the X5-2, X4-2, and X3-2 machines respectively. Making predictions using Pandia takes a fraction of a second per placement.

Figures 1 & 10 show how the predictions compare with the measured performance on the X5-2 machine. The different thread placements are indicated on the x-axis, with the placements sorted first by the total number of threads, then by the number of threads on core 0, then core 1 and so on up to core 35. The y-axis is the performance of that placement normalized to the best performance seen for that workload.

For most workloads, the measured and predicted results are visually close. Comparing the performance difference between the fastest predicted placement and the fastest tested placement we find that the mean differences are 2.8%, 0.29%, and 0.77% for the X5-2, X4-2, and X3-2 machines respectively, and the median differences are 1.05%, 0.00%, and 0.00%.

We quantify the error across all the runs in these predictions using two metrics (Figure 11a–b):

- Error: The absolute difference between the predicted and measured performance as a percentage of the measured value.
- Offset error: The mean difference between the two sets of values is added to the predicted line before measuring the absolute difference. This technique removes errors that are introduced when the two lines are some constant distance apart, so providing a measure of how accurate the output is at predicting performance trends (if not exact values).

Under these metrics, the median error across the runs is 8.5% and the median offset error is 3.6% for the X5-2, 3.8% and 1.4% for the X4-2, and 3.8% and 1.5% for the X3-2.

To test the portability of the workload descriptions between different machines we used the X3-2 workload descriptions with the X5-2 machine description and vice-versa. The resulting errors for these machines can be seen in Figure 11c & 11d respectively. These show while the relative error increases, the results still appear useful.

These are parallel benchmarks and so they tend to scale well. However, as the machines get larger, the point of peak performance become less likely to be the maximum thread count. 9% of the applications on the X4-2 do not use the maximum number of threads rising to 81% of the applications on the X5-2. The database operation Sort Join has peak performance with just 32 threads on the X5-2.

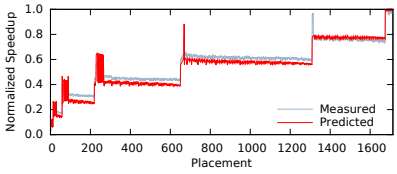
6.2 Four-socket machine

To explore the effect of systems with larger numbers of sockets we ran a series of experiments using a 4 socket Westmere machine (model X2-4). This machine has 10 cores per socket, and 80 hardware threads in total. We omit the sort-join test here because it uses AVX instructions which are not present in these older processors.

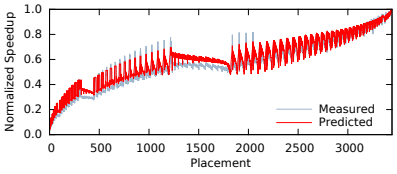
We split the different placements into three classes: those in which at most two sockets are active (providing 20 cores in total), those in which at most 20 cores are active (potentially placed over all sockets), and finally the complete set of placements over the machine. Including the 20-core runs helps us separate the move to larger numbers of sockets from the fact that runs over the complete machine will generally use more cores in total, and may show differences in behavior as a consequence

The resultant error from these runs can be seen in Figure 12. This is an older machine without adaptive caches, and we do see larger errors in the 2-socket cases when com-

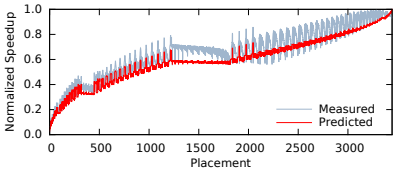
Development workloads



BT: Block tri-diagonal solver (NPB)

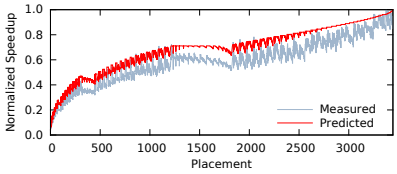


CG: Conjugate gradient (NPB)

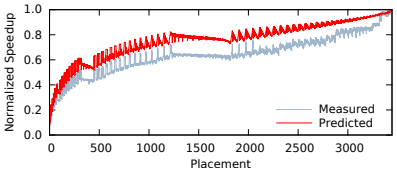


IS: Integer sort (NPB)

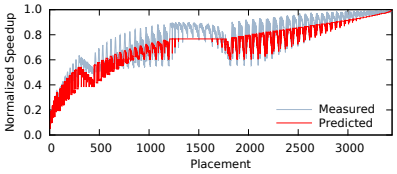
Evaluation workloads



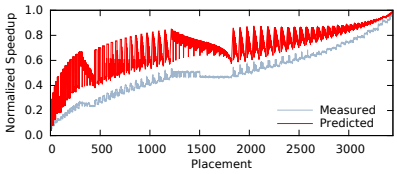
Applu: Parabolic / Elliptic PDE solver (OMP)



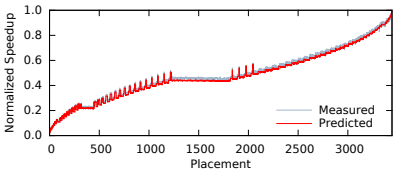
Apsi: Meteorology: pollutant distribution (OMP)



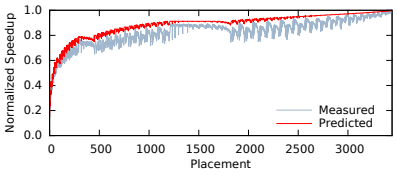
Art: Neural network simulation (OMP)



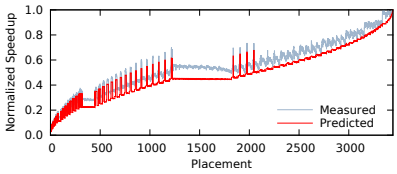
Bwaves: Blast wave simulation (OMP)



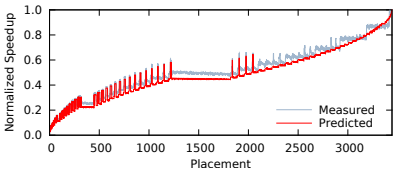
EP: Embarrassingly parallel (NPB)



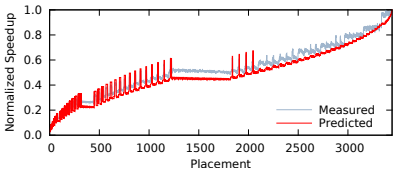
FMA-3D: Finite-element crash simulation (OMP)



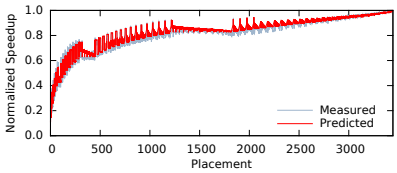
FT: Discrete 3D fast Fourier transform (NPB)



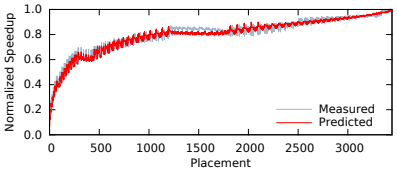
LU: Lower-upper Gauss-Seidel solver (NPB)



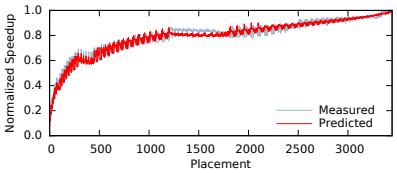
MG: Multi-grid on a sequence of meshes (NPB)



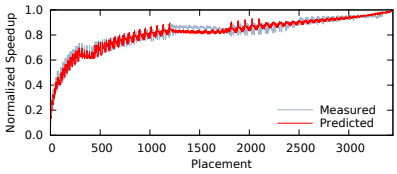
NPO: No partitioning, optimized hash join ([3])



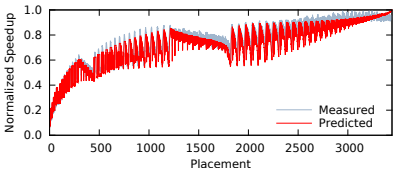
PRHO: Parallel radix histogram optimized hash join ([3])



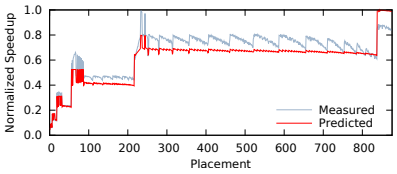
PRH: Parallel radix histogram hash join([3])



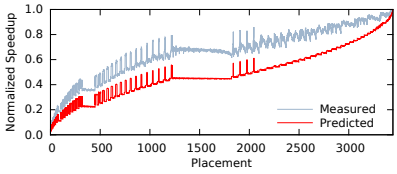
PRO: Parallel radix optimized hash join ([3])



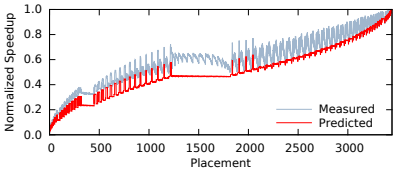
PageRank: In-memory parallel PageRank ([14])



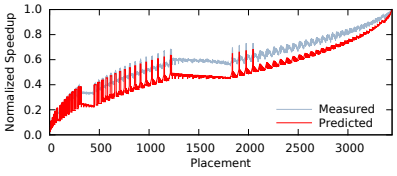
Sort-Join: In-memory sort-join ([3])



SP: Scalar Penta-diagonal solver (NPB)

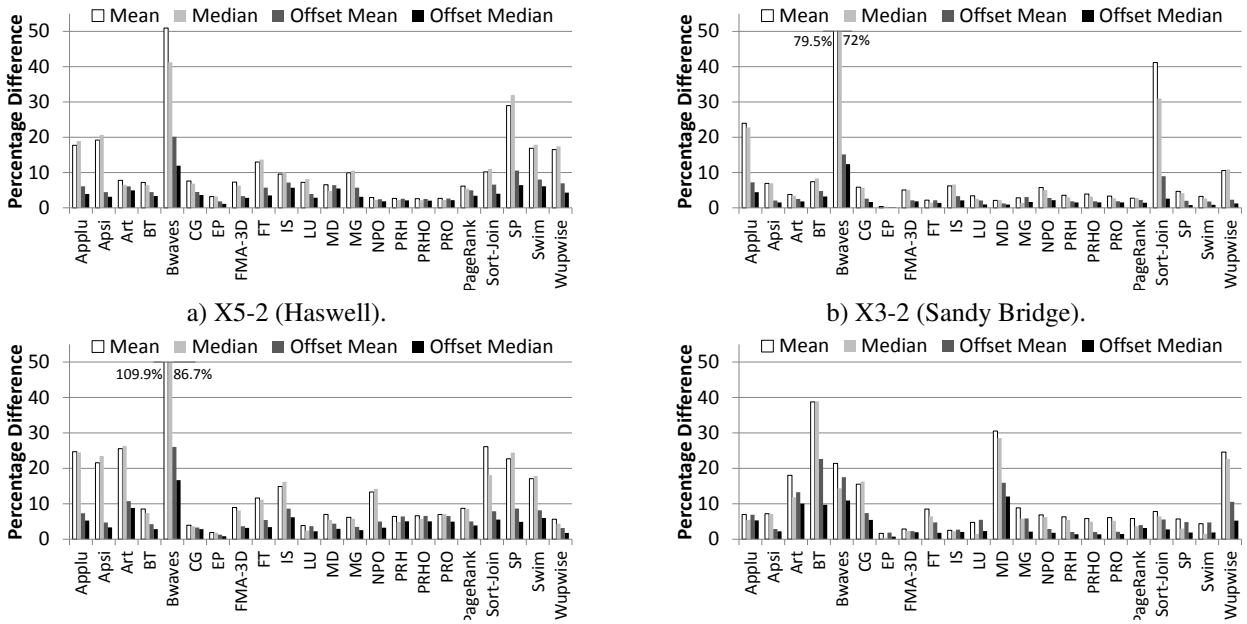


Swim: Shallow water modeling (OMP)



Wupwise: Wuppertal Wilson fermion solver (OMP)

Figure 10: Predicted versus measured performance for the benchmarks. The results for MD are shown separately in Figure 1.



a) X5-2 (Haswell).

b) X3-2 (Sandy Bridge).

c) X5-2 using X3-2 workload descriptions.

d) X3-2 using X5-2 workload descriptions.

Figure 11: Mean and median errors when predicting performance. Graph *a* (X5-2) corresponds to the results in Figures 1 and 10. Graph *b* is the errors when running on the smaller X3-2. Graphs *c* and *d* explore the portability of workload descriptions reporting the error when using the X3-2 workload descriptions on the X5-2, and the X5-2 workload descriptions on the X3-2.

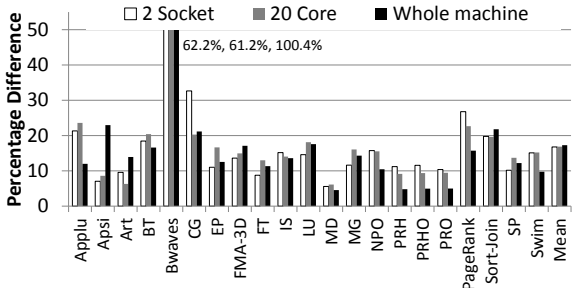


Figure 12: Mean errors on a 4 socket Westmere machine. The values here correspond to the white bars in Figure 11

paring with the newer 2-socket machines. However, we generally do not see additional errors when spreading work over the increased number of sockets.

6.3 Additional experiments

Workloads with poor scaling. To test how Pandia behaves for applications that do not scale well we ran a single-threaded version of the NPO database join (one thread is active, the others remain idle after initialization). The results can be seen in Figure 13(a). Pandia is able to detect the absence of scaling, and the impact of changes in memory placement when multi-socket placements spread the application's data over the machine.

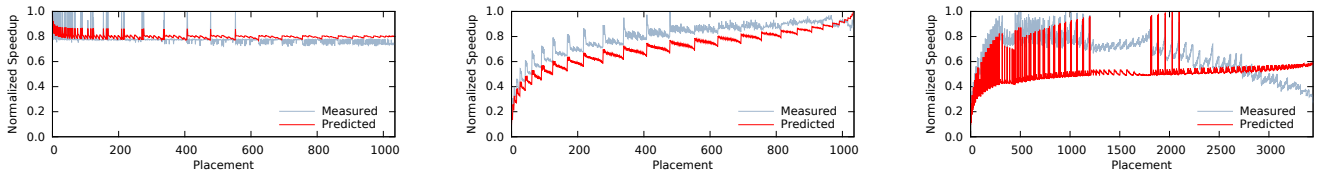
Workloads which do not follow our assumptions. Equake has a reduction step which causes the total computational demand to rise with the thread count. This violates our assumption that the total work is constant. For the 16 core X3-2 machine the predictions are good as the thread count remains relatively small (Figure 13b). For the larger 36 core X5-2 machine the impact of this broken assumption is clear (Figure 13c).

Simple pattern exploration. As an alternative to Pandia's 6 profiling runs, we considered the possibility of making a broader set of profiling runs, and simply selecting the best result from among them. Concretely, we consider a simple "sweep" of placements with $1 \ldots n$ threads placed as close together as possible on a machine, or spread as far apart.

For the X5-2 machines it took on average 8.0 times longer to explore the sweep of placements than to construct a workload description using Pandia. On the smaller X4-2 and X3-2 machines it took an average of 4.2 and 4.0 times longer to explore the placements respectively. The best placement was found by the sweep on 21 of the 22 applications for the X3-2 machines, and 20 of 22 for the X4-2 machines. On the larger X5-2 machines, the sweep found the best placement in only 8 of the 22 applications.

Although the simple sweep is effective on small machines, its effectiveness appears to diminish on larger systems. In addition, Pandia provides predictions of resource consumption as well as predictions of performance; we believe this will help make predictions when co-scheduling workloads.

Power management. Modern processors use sophisticated dynamic power management techniques. These techniques



a) Single threaded version of NPO. b) Equake on the X3-2. c) Equake on the X5-2.

Figure 13: Exploring Pandia in cases where a workload does not scale (a), and for the equake workload which does not follow our assumption that the total amount of work performed is constant as the thread count varies (b–c).

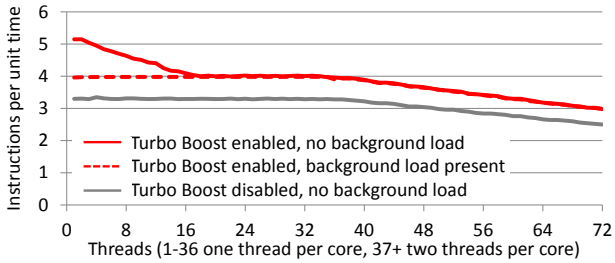


Figure 14: The effect of Turbo Boost on the rate of executing instructions in a simple CPU-bound loop on a dual-socket machine with Intel Xeon E5-2699-v3 processors. The dashed line shows the effect of enabling Turbo Boost, but running a CPU-bound workload on otherwise-idle cores.

include features such as the Turbo Boost technology in Intel processors which allows cores to run faster when only a small number of them are active.

It is common for researchers to disable these features. However, doing so is unrealistic for several reasons. First, they are generally used in production settings. Second, the performance with Turbo Boost disabled is worse than with it enabled—that is, there is a penalty for disabling this features even when all threads are active and no boost is naïvely expected. This occurs because, with Turbo Boost disabled, the chip will operate at its nominal frequency. For the X5-2, this is 2.3GHz whereas, with Turbo Boost, frequencies of 2.8GHz–3.6GHz are possible depending on the number of occupied cores [15]. Figure 14 illustrates these effects.

Our approach is to leave all hardware power management features *enabled*, but to remove these effects from our measurements by filling any otherwise-idle cores during profiling with a core-local background workload.

6.4 Limitations

Pandia currently has two principal limitations: workloads using multiple kinds of threads, and workloads with discontinuous scaling.

Multiple thread types. Many applications consist of multiple thread types, such as a master thread and $n-1$ slave threads. Currently, Pandia assumes that all threads have similar behavior. We suspect that more heterogeneous workloads could be considered by identifying groups of threads through profiling. In practice, when considering the use of our techniques in a multi-threaded server application, it may be more productive to expose thread groupings explicitly in software.

Discontinuous scaling. Pandia works on the principle that adding additional threads generally gives additional performance. However if the performance gain is discontinuous the models become less effective. An example of where this occurs is the benchmark BT with the smallest dataset size: in that case the main parallel loop has only 64 iterations followed by a barrier. By the time 32 threads are reached there will be no further performance increase until 64 threads are available. The problems inherent in such a workload are reflected in our assumption that there is sufficient fine-grained parallelism to distribute evenly.

7. Related Work

We now consider alternative approaches to predicting performance. The reasons for predicting performance vary from scheduling workloads in a machine [35], anticipating the performance of supercomputer workload before it is built [16], or extrapolating the performance of a workload being developed on smaller systems [32].

Techniques for predicting a single workload include hand built models [16], using simulators [7], rerunning traces [32], and fitting timings for runs with small numbers of threads to regression models [5]. Compared with Pandia, these techniques require more manual construction, more time to run, or are only able to handle predictions of thread count (not thread placement).

Memory and cache. Lepers *et al.* [18] describe techniques for dynamically selecting thread and memory placement to take account of asymmetries in the memory systems of NUMA machines, such as half-bandwidth or asymmetric interconnect links.

Predicting based just on cache interference can be done with mathematical models on data gathered from instrumented workloads [8, 21, 29]. McGregor *et al.* [22], Knauerhase *et al.* [17], Fedorova *et al.* [13], Zhuravlev *et al.* [34], Collins *et al.* [10] and Dhiman *et al.* [12] select threads or workloads to collocate based on bus transactions per thread, stall cycles per thread, and LLC miss rate per thread. Xie

and Loh classify applications [30] based on LLC usage and Lin *et al.* [19] on L2 usage. Their work uses a fixed number of threads, and either responds dynamically (i.e., reacting to bad situations, not predicting good allocations), or it predates adaptive caches [27] which can adapt to workload requirements (making cache size less significant in application performance, but also making measurements and predictions more difficult). ReSense [11] dynamically controls the number of threads, but does not identify the relationships between performance and thread placements that Pandia does.

CPU. Banikazemi *et al.* [4], and Zhang *et al.* [33] use CPU performance to identify combinations of applications that produce poor behavior. In Zhang *et al.*'s case, hundreds of application instances run across a cluster, allowing statistical approaches to identify poor performance.

ESTIMA [9] extrapolates performance predictions from counts of different types of stall in runs with low thread counts. From these it builds a per-application model by fitting equations to each type of stall and then fitting the resultant multi-dimensional space to previously measured times. They do not model different thread placements or resource demands.

Whole system. Q-Clouds [25] identifies interference with application level indications requiring modification to the workload. Merkel *et al.* [23] introduce "task activity vectors" to characterize current resource demands for a fixed number of threads based on performance counters.

Bhadauria and McKee [6] used performance counters to identify when programs fail to scale, and search using heuristics to choose which programs to run together, and to set thread counts. Yasin [31] describes a combination of existing and proposed hardware counters to identify the resources that constrain an application. Scal-Tool [28], uses performance counters to empirically model cache space, load imbalance, and synchronization. Scal-Tool helps explain the performance characteristics seen in a workload, rather than predicting performance. It is similar to Pandia, but predates multi-core processors, multi-threaded processors, and adaptive caches.

Mainstream operating systems use heuristics to select thread placements (for instance, always packing threads together, or always distributing threads onto different sockets). They do not set the number of software threads used by applications. Lozi *et al.* illustrate unexpected interactions can result in poor performance [20].

8. Conclusion

We have presented Pandia, a system for predicting the performance of in-memory parallel workloads from just six profiling runs. Testing on a set of 22 workloads has shown results with a median difference of 1.04% between the fastest predicted placement and the fastest measured placement, an median error of 8%. This suggests that the results can be used to make real decisions about the placements of workloads.

The model is built around measuring the CPU and bandwidth resource demands, coupled with measurements of the interactions between threads. We believe this resource-based approach will let Pandia handle mixes of workloads running together by looking at their total demands.

Perhaps surprisingly, the simple level of detail in our machine models is effective for our workloads. The trend appears to be that while individual cache architectures are becoming more complex, the necessity to model them in detail is being diminished. One reason for this difference is that hardware is now more effective in avoiding pathologically bad behavior. This makes workloads less susceptible to "performance cliffs". This change makes relatively simple models possible, but also renders many traditional models infeasible as collocated stress workloads for example to detect the required cache size of an application will no longer function without detailed knowledge of the cache implementation.

Future work. While we saw some degree of portability of workload descriptions between machines, this performs less well when going from a lower-specification machine to a higher-specification machine. This is because the initial single-thread resource demands will reflect the maximum performance of resources in the lower-specification machine, and we cannot identify how the demands will change as limits are removed. The ESTIMA techniques of Chatzopoulos may help here [9].

Pandia could also be integrated into runtime systems to choose the placement of threads in parallel loops. In this scenario the workload description could be generated during the execution of early iterations of the loop.

Finally, we aim to extend Pandia from scheduling a single workload on a single machine to the scheduling of multiple workloads on a rack-scale system. We believe Pandia's prediction of resource consumption as well as overall workload performance will let us handle cases with multiple workloads sharing a machine. As we start to consider more complex workloads, we aim to relax our assumption that workloads do not perform significant I/O—it may be that off-machine communication links can be accommodated directly in our machine models in terms of available bandwidth or I/O operation rates.

Acknowledgments

We would like to thank Cagri Balkesen, Felix Kaser, and Martin Maas for their help with benchmarks and experiments, and Alex Kogan, Dave Dice, David Vengerov, Davide Bartolini, Evgenij Belikov, Felix Kaser, Iraklis Psaroudakis, Matthew Pugh, Stefan Kaestle, and Stuart Wray, for their help and feedback in the construction of this paper. We would also like to thank the anonymous reviewers for their comments on the paper, and our shepherd Dejan Kostić for his help in preparing the camera-ready version.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks; summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165. ACM, 1991.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373, 2013.

[4] M. Banikazemi, D. Poff, and B. Abali. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the International Conference on Supercomputing*, pages 39:1–39:12, 2008.

[5] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 368–377. ACM, 2008.

[6] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th International Conference on Supercomputing*, pages 189–199. ACM, 2010.

[7] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, Feb. 2006.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[9] G. Chatzopoulos, A. Dragojević, and R. Guerraoui. ESTIMA: Extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.

[10] A. Collins, T. Harris, M. Cole, and C. Fensch. LIRA: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, pages 2:1–2:8. ACM, 2015.

[11] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Transactions on Architecture and Code Optimization*, 10(4):41:1–41:25, Dec 2013.

[12] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: A system for energy efficient computing in virtualized environments. In *Proceedings of the 14th International Symposium on Low Power Electronics and Design*, pages 243–248. ACM, 2009.

[13] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38. IEEE, 2007.

[14] T. Harris and S. Kaestle. Callisto-RTS: Fine-grain parallel loops. In *2015 USENIX Annual Technical Conference, USENIX ATC '15*, pages 45–56, July 2015.

[15] Intel Corp. *Intel Xeon Processor E5 v3 Product Family—Processor Specification Update*. 2016. `http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-spec-update.pdf`.

[16] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 37–37. ACM, 2001.

[17] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[18] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference, USENIX ATC '15*, pages 277–289, July 2015.

[19] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Conference on High-Performance Computer Architecture, HPCA-14 '08*, pages 367–378, 2008.

[20] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16. ACM, 2016.

[21] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Performance Evaluation Review*, 32(1):2–13, June 2004.

[22] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.

[23] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, pages 153–166. ACM, 2010.

[24] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran. *SPEC OMP2012 — An Application Benchmark Suite for Parallel Systems Using OpenMP*, pages 223–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[25] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds.

In *Proceedings of the 5th European Conference on Computer Systems*, pages 237–250. ACM, 2010.

[26] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News*, 35(2):381–391, June 2007.

[28] Y. Solihin, V. Lam, and J. Torrellas. Scal-Tool: Pinpointing and quantifying scalability bottlenecks in DSM multiprocessors. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99. ACM, 1999.

[29] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *SIGOPS Operating Systems Review*, 44(4):19–29, Dec. 2010.

[30] Y. Xie and G. H. Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings 2nd Workshop on CMP Memory Systems and Interconnects (CMP-MSI)*, June 2008.

[31] A. Yasin. A top-down method for performance analysis and counters architecture. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 0:35–44, 2014.

[32] J. Zhai, W. Chen, and W. Zheng. PHANTOM: Predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 305–314. ACM, 2010.

[33] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–142. ACM, 2010.

[35] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):4, 2012.