Evaluating Execution Time Predictability of Task-Based Programs on Multi-Core Processors

Thomas Grass[1,2], Alejandro Rico[2], Marc Casas[2], Miquel Moreto[1,2], and Alex Ramirez[1,2]

[1] Universitat Politècnica de Catalunya, 08034, Barcelona, Spain
[2] Barcelona Supercomputing Center, 08034, Barcelona, Spain

Abstract. Task-based programming models are becoming increasingly important, as they can reduce the synchronization costs of parallel programs on multi-cores. Instances of the same task type in task-based programs consist of the same code, which leads us to the hypothesis that their performance should be regular and thus their execution time should be predictable. We evaluate this hypothesis for a set of 12 task-based programs on 4 different machines: a high-end Intel SandyBridge, an IBM POWER7, an ARM Cortex-A9 and an ARM Cortex-A15. We show, that predicting execution time assuming performance regularity can lead to errors of up to 92%. We identify and analyze three sources of execution time impredictability: input dependence, multiple behaviors per task type and resource sharing. We present two models based on linear interpolation and clustering, reducing the prediction error to less than 12% for input dependent task types and to less than 2% for task types with multiple classes of behavior. All in all, this work invalidates the assumption that performance is always regular across instances of the same task type and quantifies its variability on a wide range of benchmarks and multi-core systems.

Keywords: Execution Time Predictability, Task-Based Programming Models, Multi-Core.

1 Introduction

Multi-core systems are integrating an increasing number of processor cores on a single chip. This makes it difficult for programmers to exploit the available on-chip thread-level parallelism.

Task-based programming models allow the programmer to specify program parts called *tasks*. Tasks may execute concurrently and are typically instantiated many times during execution. A runtime environment dynamically maps task instances to threads. The intuitive program partitioning improves programmability. At the same time, dynamic task scheduling reduces the inherent synchronization costs of other shared memory programming models thanks to a better load balancing [1].

The fact that all instances of the same task type consist of the same static code suggests that they should exhibit similar performance and execution time and, therefore, execution time should be predictable. In this paper, we investigate the execution time predictability of task-based programs based on performance regularity. We carry out an analysis on four different state-of-the-art multi-core machines, two based on ARM Cortex-A9 MPCore and Cortex-A15 MPCore mobile CPUs, and the other two based on high-end Intel Sandy Bridge and IBM POWER7 CPUs. This allows us to investigate if performance regularity depends on the architecture. We expect performance variability to increase when increasing the number of execution threads competing for shared resources. Therefore, we analyze performance variability on a per-task-instance basis for thread counts ranging from one to the number of cores on each machine. We reach similar conclusions for the different machines, but find that architectures with more aggressive performance optimizations show a higher performance variability.

We identify three sources of variability across instances of the same task type: input dependence, multiple classes of behavior and contention on accessing shared resources. For programs suffering from resource contention, we investigate how sharing decreases performance and increases performance variability. We also present a model based on linear interpolation to predict execution time of input dependent task types. Furthermore, we use a clustering algorithm to identify different behaviors in the same task type. Using our interpolation model and clustering algorithm, we dramatically increase the accuracy of execution time prediction. Prediction errors over 80% are reduced to less than 12% for input dependent cases and less than 2% on the presence of multiple behaviors.

The contributions of this paper are the following:

- An analysis of performance variability across instances of the same task type in task-based programs running on multi-core systems. This analysis shows the variability on an instance by instance basis.
- A classification of sources of execution time variability on instances of the same task type.
- A low-complexity model based on linear interpolation for predicting the execution time of a task instance as a function of its instruction count.
- The use of a clustering algorithm to identify different classes of behavior in the same task type. In our example, we successfully classify task instances into clusters that exhibit, each of them, regular performance.

2 Related Work

To the best of our knowledge, this is the first analysis of execution time predictability on task-based programs. However, there are other performance analyses of task-based programs focusing on other aspects.

Duran et al. [4] present a benchmark suite consisting of task-based OpenMP programs. They give examples for different kinds of performance analyses of these benchmarks. They evaluate total execution time as a function of various parameters such as processor count and task creation cut-off parameters. Other works [14,15] investigate task granularity and task creation cost as performance-limiting factors in task-based programs. However, these works neither analyze performance on a per-task-instance basis nor task execution time predictability.

There are other works that use analytical models to predict execution time [8,10,6]. These works use mathematical models to compute the delays of certain events during execution. Most past works compute delays for events at the instruction-level, such as instruction issue and commit, branch mispredictions and cache misses. Our model works at a coarser granularity by computing the delay of whole individual tasks.

Performance predictability of parallel applications on large HPC systems has been explored from many perspectives. Some approaches combine the efficiency of analytical models with the accuracy of simulation to generate accurate and fast performance predictions [16]. Other approaches [9] explore performance predictability by developing application-specific performance models, which are formulated from an analysis of the code, inspection of key data structures, and analysis of traces gathered at runtime. While this methodology provides fast and accurate predictions, it is application specific and it requires a deep understanding of the scientific codes. These works target MPI applications while the work in this paper focuses on shared memory task-based programs.

3 Execution Time Predictability of Task-Based Programs

Many parallel implementations of numerical algorithms decompose the problem domain into sub-domains called *blocks* or *tiles*. In task-based programming models the programmer specifies parts of a program as work units called *tasks*, each one to perform a different operation. A task is usually instantiated many times, each instance performing the common operation of the task on a separate block or tile. Task instances can be scheduled to threads whenever they have their dependencies satisfied. Typically, a thread executes many task instances before reaching a synchronization point.

The fact that instances of the same task type consist of the same code leads us to the assumption that they consist of similar numbers of instructions, exhibit similar performance and therefore their execution time is predictable. However, this assumption turns out to be wrong in some cases. Fig. 1 shows the total execution time prediction error for a set of task based programs, assuming the time of the first or the second executed instance for all instances of a task type. The error is calculated according to Eqn. 1, with T the set of task instances of the same task type, C_{Sample} the cycle count of the sample task instance and C_i the cycle count of task instance i. We only investigate time spent in task execution and ignore operating system and runtime system overheads.

$$Err = \left(1 - \left|\frac{\sum_{i \in T} C_{Sample}}{\sum_{i \in T} C_i}\right|\right) \cdot 100\% \tag{1}$$

Before conducting our detailed analysis, we envision three potential sources of performance variability that potentially degrade performance predictability:

- *Input dependence*: The behavior of a task instance is input dependent. An example is sparse algorithms in which task instances perform different amounts of computation or exhibit different memory access patterns.

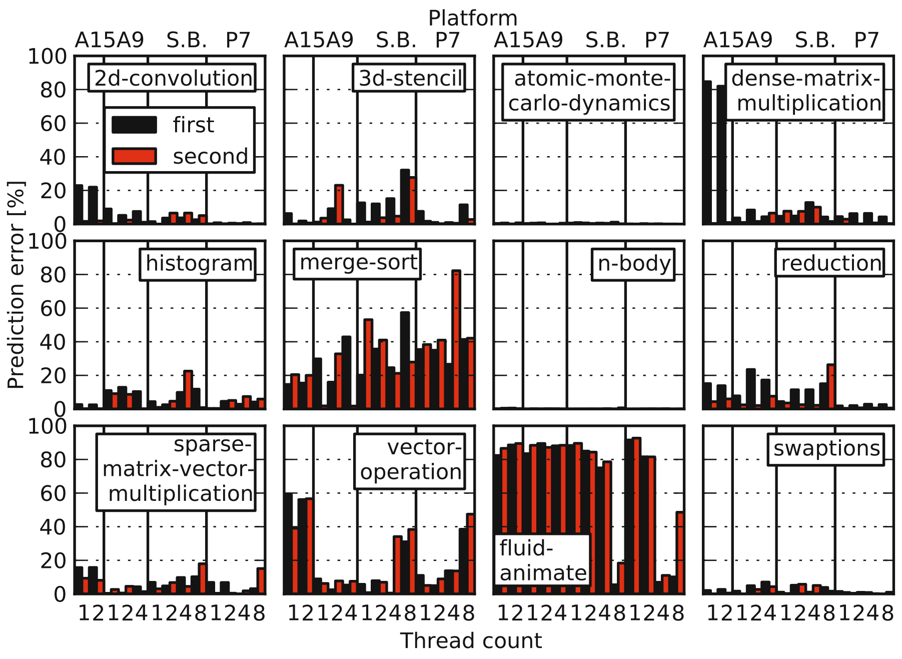

Fig. 1. Percent error when assuming the execution time of the first / second executed task instance for all task instances to predict total execution time. Results shown for four different machines (see Tab. 2) and different thread counts.

– *Several types of behavior per task type*: Task instances of the same type perform one out of several possible types of computation. An example is recursive algorithms in which some task instances create more child tasks, while others perform the actual computation when the recursion terminates.
– *Shared resources contention*: Multiple threads interfere with each other when accessing shared resources. Different instances of the same task type may suffer from different degrees of interference caused by other threads running in the system and accessing shared resources. This includes shared caches, interconnect structures and memory bandwidth.

4 Experimental Setup

In this section we present the experimental setup used for the performance analysis in this paper. First, we give a brief overview of OmpSs [5], the task-based programming model used for our analysis. Afterwards, we explain how we measure the performance of OmpSs programs on a per-task basis. We present the investigated benchmarks and explain how we configured them to obtain meaningful results. Finally, we present the platforms on which we run our experiments.

Programming Model: OmpSs is an extension of OpenMP 3.0. It consists of the Mercurium compiler and the NANOS++ runtime environment. In addition to the features of OpenMP 3.0, it allows to annotate tasks with data inputs and outputs. The NANOS++ runtime system automatically manages inter-task data dependencies and schedules and synchronizes task instances accordingly. These OmpSs features were included in the recent OpenMP 4.0 specification.

Measuring the Performance of Tasks: We measure cycle count, instruction count and numbers of L1 (data), L2 (data) and L3 cache misses using hardware performance counters. We modify Mercurium to insert calls to a low-overhead instrumentation library at the beginning and the end of each task instance. This instrumentation library is an interface to the PAPI library [3]. Since NANOS++ can suspend a task instance before it completes, we also instrument NANOS++ to pause the performance measurement if a task is suspended.

Benchmarks: In this paper we investigate a set of 12 parallel benchmarks. They are task-based programs implemented in the OmpSs programming model. The benchmarks and their key characteristics are listed in Tab. 1. They cover a wide range of algorithms that are widely used in HPC scientific applications and include programs with different compute-to-memory ratios, different memory access patterns and different amounts of parallelism and synchronization. The first ten benchmarks have been successfully used in previous works to evaluate HPC clusters [12,13], while *fluidanimate* and *swaptions* are part of the PARSEC benchmark suite [2]. As we conduct this work, these are the only benchmarks of the PARSEC suite for which there is an OmpSs implementation available. We perform ten executions of each benchmark for each configuration and choose the fastest one for our evaluation to minimize OS noise.

Application Tuning: We classified the benchmarks according to whether they are compute-intensive or not. Because the working sets of all concurrently executing task instances fit into the last level cache, we considered the following benchmarks as compute-intensive: *2d-convolution*, *3d-stencil*, *atomic-monte-carlo-dynamics*, *merge-sort*, *dense-matrix-multiplication*, *fluidanimate* and *swaptions*.

We optimized compute-intensive benchmarks by adjusting the task working set to fit into the on-chip last-level cache. This is one of the most straightforward optimizations applied by programmers in blocked numerical algorithms. The most cache constrained configuration is the Cortex-A9 running with four threads. Therefore, we adjusted the task working set to fit a fourth of the last-level cache in the Cortex-A9 chip. We use the same configuration for all platforms to have the same basis for comparison.

For the remaining benchmarks, we configured the task working set for the resulting task instances to be at least 100 000 instructions long. By doing so, we ensure that the time spent in task execution is significantly larger than the time spent in performance measurement code. The number of task instances per

Table 1. Investigated benchmarks

Benchmark	**Properties**
2d-convolution	Strided accesses
3d-stencil	Strided accesses
atomic-monte-carlo-dynamics	Embarrassingly parallel
dense-matrix-multiplication	High data reuse, compute bound
histogram	Atomic operations
merge-sort	Recursion, many synchronizations
n-body	Irregular memory accesses
reduction	Parallelism decreases over time
sparse-matrix-vector-multiplication	Load imbalance, memory bound
vector-operation	Regular, memory bound
fluidanimate	Variable task instance size
swaptions	Regular, computation bound

Table 2. Investigated machines

Micro-arch.	**Cores**	**L1 size**	**L2 size**	**L3 size**	**Memory**
ARM Cortex-A15 MPCore	2	32KB+32KB per core	1MB shared	n/a	2GB 32-bit DDR3L-1600
ARM Cortex-A9 MPCore	4	32KB+32KB per core	1MB shared	n/a	2GB 32-bit DDR3L-1500[1]
Intel Sandy Bridge	8	32KB+32KB per core	256KB per core	20MB shared	32GB 64-bit DDR3-1600
IBM POWER7	8	32KB+32KB per core	256KB per core	32MB shared	64GB 64-bit DDR3-1600

application is adjusted to a large enough number so there is enough parallelism to use all threads at all times.

Investigated Platforms: Tab. 2 gives an overview of the characteristics of the four machines used for the evaluation in this paper. The first two platforms are based on low-power mobile systems-on-a-chip, while the other two are high-end machines used in HPC environments. This selection of machines covers three of the most important ISAs nowadays: x86-64, POWER ISA, and ARMv7. Even

[1] DDR3L-1600 connected to a 750MHz interface.

though ARM microprocessors are not used in HPC environments yet, there is an increasing interest in integrating ARM chips in future server and HPC machines [7,13]. Besides, these four machines cover a wide range of performance levels as well as different ISAs, CPU, cache and memory technologies.

5 Evaluation

The results of the experiments conducted in the scope of this paper show that, despite the obvious intuition, performance can be irregular across the instances of the same task type. This directly affects execution time prediction (shown in Fig. 1). In this section, we first show the results of our performance analysis on a per-task-instance basis. Afterwards, we present a case of input dependent task behavior and present a model to estimate the execution time of a task instance as a function of its instruction count. We also show a case of multiple classes of behavior within a single task type. We use a clustering technique to distinguish different classes of behavior and improve execution time predictability. Finally, we explain how resource sharing affects performance regularity and analyze contention on different resources in the memory hierarchy.

5.1 Per-Task-Instance Performance Analysis

Fig. 2 shows boxplots of the measured instructions per cycle (IPC) per task type. Each chart corresponds to one task type and shows the measured results on four different platforms. Only one thread per core is executed in each experiment, which limits the configurations to two threads (Cortex-A15), four threads (Cortex-A9), and eight threads (Intel Sandy Bridge and IBM POWER7). The solid box contains the interquartile range of the measured IPC values of all instances of the respective task type, i.e., 50% of the observations are within this range. The horizontal line within the box indicates the median. The whiskers extend from the 5th to the 95th percentile. The lower and upper 5% of the measured IPC values are treated as outliers.

Most of the investigated benchmarks only have one task type, whereas *merge-sort*, *n-body* and *reduction* have two and *fluidanimate* has eight. The different task types of *fluidanimate* show similar performance variability. Therefore, we limit our evaluations to the task type ComputeForcesMT which accounts for 40% of *fluidanimate*'s total instruction count.

In our results we observe two general classes of behavior. The first class consists of benchmarks for which IPC does not significantly degrade when increasing the number of execution threads. This behaviour is exposed by the benchmarks *2d-convolution*, *atomic-monte-carlo-dynamics*, *merge-sort* (both tasks), *n-body* (both tasks), *reduction* (both tasks), *fluidanimate* (all task types) and *swaptions.* We make the important observation that *2-d convolution*, *atomic-monte-carlo-dynamics* and *n-body* (task type 1) present a nearly constant IPC with very low variability. This behavior is persistent across the different platforms.

The second class of behavior consists of the benchmarks for which IPC degrades when increasing the number of execution threads. This phenomenon is

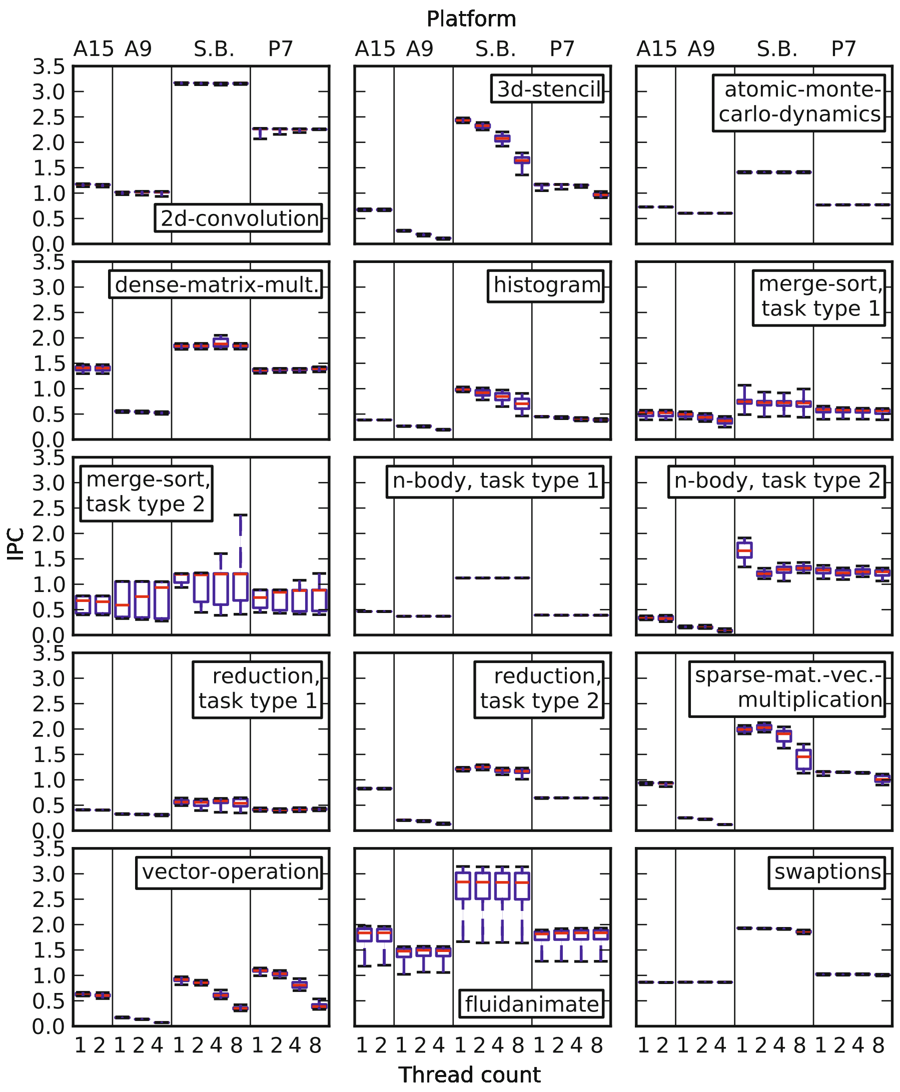

Fig. 2. IPC variation per task type on four different platforms (ARM Cortex-A9 and A15, Intel Sandy Bridge and IBM POWER7)

known as work time inflation [11]. In our benchmark suite, this behavior is exposed by the benchmarks *3d-stencil*, *histogram*, *sparse-matrix-vector-multiplication* and *vector-operation*. For these benchmarks, we also observe an increasing performance variability. Note that the variability shown in Fig. 2 directly relates to the prediction error shown in Fig. 1.

5.2 Predictability of Irregular Behavior

In this subsection we identify three sources of irregular behavior, namely input dependence, multiple classes of behavior per task type and resource sharing. We predict execution time of task types with input dependent behavior using an interpolation-based model. For task types with several classes of behavior we use a clustering algorithm to detect clusters of similar behavior and predict execution time on a per-cluster basis. Finally, we analyze the impact of resource sharing on performance predictability.

Input Dependence: Input dependence is the dependence of the control flow of a task instance on the input data. Fig. 3 shows heatmaps of the programs *fluidanimate* and *merge-sort*. Heatmaps are a representation of a two-dimensional histogram. The colours indicate how many task instances have a certain instruction count and a certain IPC.

In the case of *fluidanimate*, the instruction count of task instances varies between 1 million and 70 million instructions, while IPC tends to be higher for higher instruction counts. This results in different numbers of execution cycles. Assuming the same cycle count for all task instances leads to the prediction error shown in Fig. 1 which reaches over 80%. The instruction count and IPC variation is caused by the fact that all task instances perform an index computation that is highly inefficient for high indexes. We want to emphasize that this index computation is part of the default implementation of the *fluidanimate* benchmark and is not caused by porting the benchmark to the OmpSs programming model.

For the programs *fluidanimate* and *merge-sort* (task type 1) we apply a sampling-based model to predict execution time as a function of instruction count for all task instances. This model assumes that the instruction count of each task instance is known apriori and works as follows. First, we add instruction count and execution time of the first executed task instance to the (empty) set of support points. Afterwards, for each encountered task instance we check if its instruction count is less than 90% of the smallest or more than 110% of the largest instruction count in the set of support points. If this is the case, we add it to the set of support points. Otherwise, we predict the execution time by linear interpolation within the set of support points or by constant extrapolation in the range outside the support points. Fig. 4 shows that the error of the total execution time prediction based on this model stays below 12% for all configurations on the Intel Sandy Bridge machine.

Multiple Behaviors Per Task Type: For *merge-sort* (task type 2) we observe two clusters in the heatmap plot, indicating two different behaviors. Strictly speaking, this is also a case of input dependence. However, the difference to the type of input dependence covered in the previous section is that there are multiple classes of behavior. This is caused by the recursive implementation of the merge sort algorithm. A task instance either creates two child instances, resulting in the cluster on the left, or it performs a sorting operation, resulting in the cluster on the right. Predicting execution time based on the assumption of regular execution time and IPC leads to the error shown in Fig. 1.

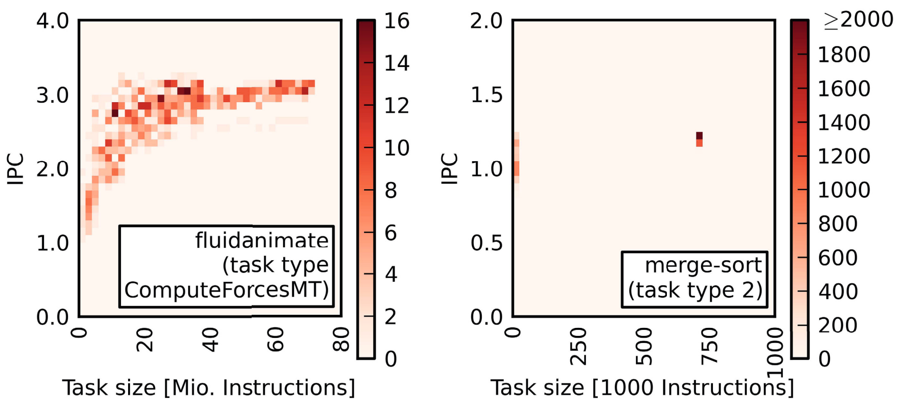

Fig. 3. Instruction count vs. IPC histogram of benchmarks *fluidanimate* (task type ComputeForcesMT) and *merge-sort* (task type 2)

For the aforementioned case, we perform a k-means clustering of all task instances into two clusters, according to their instruction count. For each cluster, we determine the centroid and chose the task instance closest to the centroid as a representative of the respective cluster. Finally, we estimate the total execution time of each cluster by multiplying the execution time of the representative by the number of task instances in the cluster. Fig. 4 shows that the error of the total execution time prediction based on this method is smaller than 2% for all configurations on the Intel Sandy Bridge machine.

Resource Sharing: The third source of irregular behavior we identified is resource sharing. In the following, we present four examples of resource sharing. These examples have in common that contention on shared resources affects the performance of task instances of the same task type to a different extent. This increases performance variability and thus decreases performance predictability. Fig. 5 shows boxplots of L2 data cache and L3 cache misses per 1000 executed instructions (misses per kilo-instruction, MPKI) of the benchmarks for which we observed a decrease of IPC for increasing thread counts. The measured number of L3 cache misses includes misses caused by L2 data cache misses due to the limitations of the available hardware performance counters.

For *3d-stencil* we observe an increase of L2 MPKI when increasing the number of threads. However, L3 MPKI stays nearly constantly low. Our theory is that the increased L2 MPKI is caused by invalidations of data residing in the private L2 caches by other threads.

The *histogram* benchmark shows not only an increase of L2 MPKI for increasing thread counts, but also an increase in L2 MPKI variability. For increasing thread counts there might be several threads competing to execute an atomic operation, resulting in higher contention. Furthermore, the execution of the atomic operation itself can invalidate data in other threads' private caches.

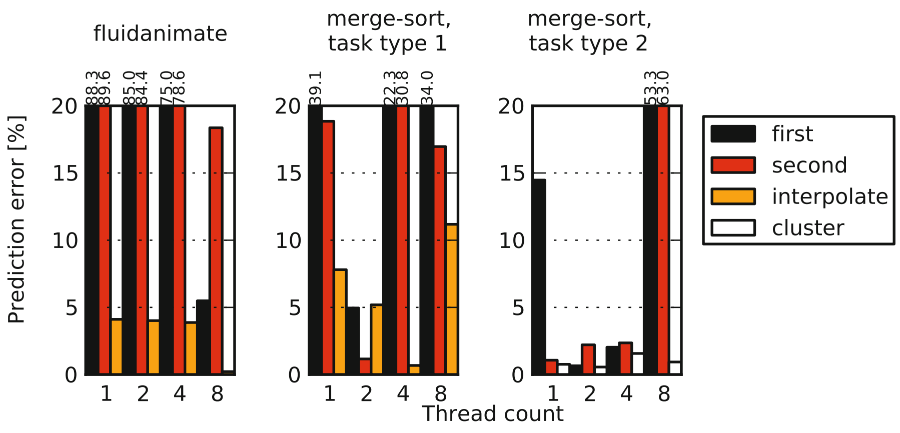

Fig. 4. Execution time prediction error using interpolation model (*fluidanimate* and *merge-sort*, task type 1) and clustering (*merge-sort*, task type 2)

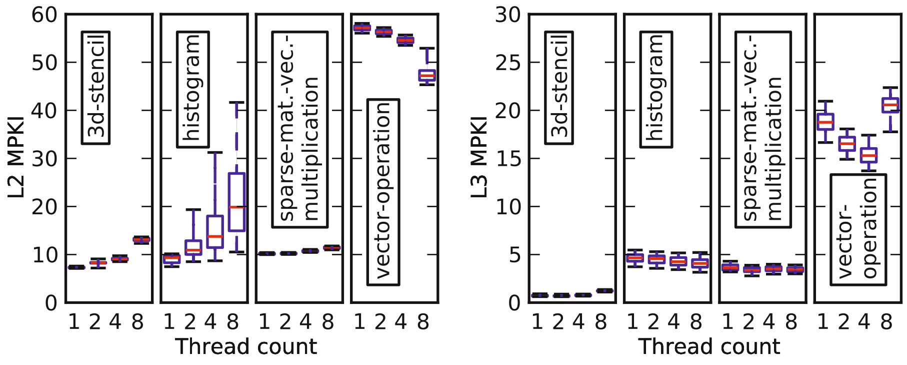

Fig. 5. L2 data and L3 cache misses per 1000 instructions (MPKI) for *3d-stencil*, *histogram*, *sparse-matrix-vector-multiplication* and *vector-operation*, executed on Intel Sandy Bridge with 1, 2, 4 and 8 threads

In case of *sparse-matrix-vector-multiplication*, L2 MPKI and L3 MPKI are nearly constant for increasing thread counts. Since the benchmark does not use shared data, the decrease in IPC has to occur due to the limited capacity of shared resources, e.g. memory bandwidth or cache bandwidth.

For *vector-operation* we observe a decrease of L2 MPKI when increasing the number of execution threads. As memory bandwidth saturates for increasing thread counts, threads progress at a slower rate and thus cause less demand misses in the L2 cache.

6 Conclusions and Future Work

The analysis in this paper shows that the naive assumption of regular performance within a task type is not always valid. However, we show that accurate performance predictions can be derived from detailed performance information of a relatively small number of task instances.

We present techniques to improve prediction accuracy for task types with irregular performance. These techniques are based on linear interpolation and clustering. The prediction error is reduced from over 80% to less than 12% for input dependent cases and less than 2% when having multiple classes of behavior. Further research is needed to improve execution time predictability of task-based programs experiencing contention on shared resources.

We envision a potential application of the insights in this paper in the fields of multi-core architecture simulation and dynamic task scheduling on multi-cores. If the performance of a task type is predictable it is only necessary to simulate a subset of all task instances, and smart scheduling techniques can be applied with apriori-knowledge of the execution time of a task instance.

References

1. Amarasinghe, S., et al.: ASCR programming challenged for exascale computing. Report of the 2011 Workshop on Exascale Programming Challenges (2011)
2. Bienia, C., et al.: Benchmarking Modern Multiprocessors. PhD thesis, Princeton University (January 2011)
3. Browne, S., et al.: A portable programming interface for performance evaluation on modern processors. Journal of High Performance Computing Applications 14(3), 189–204 (2000)
4. Duran, A., et al.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: ICPP 2009, pp. 124–131 (2009)
5. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02), 173–193 (2011)
6. Genbrugge, D., et al.: Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In: HPCA 2010, pp. 1–12 (2010)
7. Halfhill, T.R.: ARM's 64-Bit Makeover. Microprocessor Report (December 24, 2012)
8. Karkhanis, T.S., et al.: A first-order superscalar processor model. In: ISCA, Washington, DC, USA, p. 338 (2004)
9. Kerbyson, D.J., et al.: Predictive Performance and Scalability Modeling of a Large-scale Application. In: Supercomputing 2001, p. 37 (2001)
10. Nussbaum, S., et al.: Modeling superscalar processors via statistical simulation. In: PACT, pp. 15–24 (2001)
11. Olivier, S.L., et al.: Characterizing and mitigating work time inflation in task parallel programs. In: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12. IEEE (2012)
12. Rajovic, N., et al.: Experiences with Mobile Processors for Energy Efficient HPC. In: DATE 2013, pp. 464–468 (2013)
13. Rajovic, N., et al.: Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In: SC 2013 (2013)
14. Rico, A., et al.: Available Task-level Parallelism on the Cell BE. Scientific Programming 17(1-2), 59–76 (2009)
15. Schmidl, D., Philippen, P., Lorenz, D., Rössel, C., Geimer, M., an Mey, D., Mohr, B., Wolf, F.: Performance analysis techniques for task-based openMP applications. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 196–209. Springer, Heidelberg (2012)
16. Snavely, A., et al.: A Framework for Performance Modeling and Prediction. In: Supercomputing 2002, pp. 1–17 (2002)