Parallelism Profiling and Wall-time Prediction for Multi-threaded Applications

Achille Peternier
Institute of
Computational Science (ICS)
University of Lugano (USI)
Lugano, Switzerland
achille.peternier@usi.ch

Walter Binder
Akira Yokokawa
Dynamic Analysis Group (DAG)
University of Lugano (USI)
Lugano, Switzerland
{first.last}@usi.ch

Lydia Chen
IBM Research Lab
Zurich, Switzerland
YIC@zurich.ibm.com

ABSTRACT

A detailed and accurate characterization of the parallelism of applications is essential for predicting their wall-time on different platforms, both for an application running in isolation and for a set of consolidated applications executing on the same platform. However, prevailing profilers are often based on sampling and do not provide exact information on the parallelism of the profiled application. In this paper we present a novel profiler that logs all thread scheduling activities within the operating system kernel. These logs enable us to accurately characterize applications' parallelism on a given platform by computing the number of threads that are active at each moment. We also present a simple mathematical prediction model to estimate wall-time for program execution on a k_2-core machine using profiles collected using a k_1-core machine (of the same architecture and running at the same clock speed). We use our profiler to assess the parallelism of several CPU-bound DaCapo benchmarks and evaluate the accuracy of our prediction model.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

Keywords

Profiling, workload characterization, performance prediction, multicores

1. INTRODUCTION

The level of parallelism is a crucial parameter in determining applications' performance, especially in the multicore era. An application with high level of parallelism is able to leverage the abundant hardware resource available in today's computing system, particulary the CPU cores. However, many applications have limited parallelism, which can potentially result in system under-utilization. The straightforward solution to this problem is to consolidate such applications on a single system. The challenge here is how to allocate resources to those consolidated applications based on their level of parallelism, such that the overall system efficiency is maximized.

As today's applications are built on a complex software stack, it is not a trivial task to know the parallelism level of applications, especially when runtime systems of managed languages—which often feature concurrent dynamic code optimization and parallelized Garbage Collection (GC)—are part of the stack. To predict the wall-time of an application via its parallelism level, profiling methods usually require the execution of applications on all system configurations of interest. The complexity of profiling thus grows in the number of applications as well as system configurations. In this paper, we seek a light-weight profiler, which only collects per-application parallelism on a base system with k_1 CPU cores and enables the wall-time prediction on different systems with k_2 cores. We will assume that the k_1-core and k_2-core systems only differ in their number of cores, but otherwise feature the same architecture and have the same hardware and software components. That is, our parallelism characterization and wall-time predictions are platform-specific.

In this work we first present a novel profiler which operates at the Linux-kernel level and offers detailed and configurable traces of scheduler behavior. Based on these traces, a detailed characterization of the parallelism of applications is possible. Second, we provide a simple wall-time prediction model for (multi-threaded) applications on k-core machines using our parallelism characterization. Third, we present a detailed parallelism characterization of selected CPU-bound benchmarks of the DaCapo suite [3]. Fourth, we predict the wall-time of these benchmarks on a multicore machine with a varying number of cores and assess the relative error of our predictions.

2. KERNEL-LEVEL PARALLELISM PROFILER

Our approach is based on empirical analysis and measurements. We aim at minimizing the perturbations introduced by monitoring the applications we are profiling. To this end, we implemented a kernel-level module based on [12] that keeps track of all the state changes happening within a specified list of threads defined by their process and thread identifiers (Linux PIDs and TIDs). Each time a state change happens, the modified kernel scheduler logs the correspond-

ICPE'13, March 21–24, 2013, Prague, Czech Republic.

ing event into the system log directory. This information enables *post-mortem* analysis of the parallelism of an application during its runtime.

The parallelism profiler is composed of three main elements: (1) the kernel module, (2) the command-line configurator, and (3) the trace generator.

The kernel module patches the Linux kernel by adding per-thread state logging to the methods that are invoked by the OS scheduler upon each thread state change. In this way, internal kernel-space events are logged in user-space and become accessible through log files. Each time such an event is generated, timestamp, TID, PID, and the new state are recorded.

In Linux-based systems, the thread state is encoded as R when the thread is running or runnable, as S when it is waiting for an event to complete, as T when it is stopped, etc. In our work, we are mainly interested in distinguishing the R state from all the other states to identify when and how long a thread has been *active* (i.e., running or runnable).

The command-line configurator is used to intercept all the threads that are spawned within a specified process. The configurator is built on top of *libmonitor*[1] and logs the timestamps, TIDs, and PIDs of the various threads and subprocesses starting and terminating.

The trace generator gathers the log files produced by both the kernel module and the command-line configurator to combine the information using the timestamps and to generate a trace of the state changes that happened during the observed time for each thread, including thread creation and termination events.

Thanks to the traces generated by our parallelism profiler, we can evaluate the maximum degree of parallelism that an application is using (i.e., identify how many concurrent threads are being used), and for each level of parallelism $j \geq 0$, we can compute the wall-time spent when j threads were active. Based on the information provided by the traces, parallelism profiles can be created to point out an application's parallelism changes over time. Metrics such as the level of parallelism reached within a specific time interval, or for how long a specific level of parallelism is available can then be computed to produce charts such as the ones used in our evaluation.

The total overhead added by our monitoring infrastructure depends on the state change frequency. For the DaCapo benchmarks we used in our evaluation, the overhead is 2.16% on average, with a maximum of 7.98%.

3. WALL-TIME PREDICTION MODEL

In this section, we present a predictive model for multicore systems using our profiling tool. The idea is based on Amdahl's Law [9], which states that the execution time can be improved by speeding up the parallel part of the program on additional hardware. The challenge of applying Amdahl's Law lies in identifying and quantifying the parallel part of a program. In a multi-threaded program, the parallel part can be viewed as the fraction of the program where there are more than one active threads. As the number of active threads is not constant, the degree of parallelism in a program also fluctuates. One needs to incorporate variability of parallelism in applying Amdahl's Law to predict the program execution time on a k-core system. Furthermore, depending on the number of available cores, those fractions of time can be accelerated accordingly.

To such an end, we first leverage the timing information about the number of active threads provided by our profiling tool. Let $T_j(1)$ denote the execution time when there are j active threads executing the program on a single core. The number of active threads can range from zero to J, that is, $j \in \{0, \ldots J\}$. Zero active thread denotes that no thread is active (e.g., all threads are blocked on I/O). Also let k denote the number of cores available for execution. On the one hand, when the number of cores is greater or equal to the number of threads (i.e., $k \geq j$), all j threads can run on available cores without any competition, assuming no dependencies among those threads. On the other hand, when there is not a sufficient number of cores for j threads (i.e., $j > k$), the maximum speedup is restricted by the number of available cores. As such, when executing on a k-core system, the execution under j active threads, $T_j(k)$, can be further improved by the minimum value between j threads and k cores:

$$T_0(k) = T_0(1)$$

$$T_j(k) = \frac{T_j(1)}{\min\{j, k\}}, j > 0.$$

Note that when $j = 0$ or $j = 1$, no acceleration is possibile. The overall execution time of a program on a k-core system, $E(k)$, is thus the summation of fractions of time in all possible numbers of active threads:

$$\begin{aligned} E(k) &= \sum_{j=0}^{J} T_j(k) \\ &= T_0(1) + \sum_{j=1}^{J} \frac{T_j(1)}{\min\{j, k\}} \end{aligned}$$

To accommodate wider profiling and prediction scenarios, that is, on any different number of cores, we further generalize the aforementioned analysis. The profiling of active threading time is executed on a system with k_1 cores and we aim at obtaining the prediction of execution time on a system with k_2 cores. To derive $T_j(k_2)$, we first classify two cases, (1) $k_1 \leq k_2$ and (2) $k_1 > k_2$, and then discuss the performance improvement (resp. slowdown) due to the parallelism.

In case one, various parts of execution time of a program can be improved, depending on the relationship between the degree of parallelism, $T_j(k_1)$, k_1, and k_2. When $j \leq k_1$, it implies this part of the program is at its best speed and cannot be further accelerated by being executing on k_2-cores. We thus know $T_j(k_1) = T_j(k_2)$, $j \leq k_1 \leq k_2$. As for the times when $j > k_1$, $T_j(k_1)$ can be improved by leveraging the extra cores on a k_2-core system, given the available threads and cores. Essentially, the execution time of j threads on k_2 cores is speed by a factor of $\frac{\min\{j,k_2\}}{k_1}$, from $T_j(k_1)$. The rational behind is that as $T_j(k_1)$ is already fully acerbated by k_1 cores, the improvement of $T_j(k_2)$ needs to be first normalized by k_1 and then multiplied by the maximum parallelism, that is, $\min\{j, k_2\}$. As such, one can derive $T_j(k_2) = T_j(k_1)/(\frac{\min\{j,k_2\}}{k_1})$. Similar analysis can be applied for case two. The only difference is that some part of the program can be scaled down, due to the decrease of parallelism when moving from k_1 cores down to k_2 cores.

[1] https://outreach.scidac.gov/projects/libmonitor/

In summary, the generalization of our predictive model is the following:

$$E(k_2) = \sum_{j=0}^{J} T_j(k_2),\ where$$

$$T_j(k_2) = \begin{cases} T_j(k_1), & j \le \min\{k_1,\, k_2\}; \\ T_j(k_1)\frac{k_1}{\min\{j,\, k_2\}}, & k_1 \le k_2, k_1 \le j. \\ T_j(k_1)\frac{\min\{j,\, k_1\}}{k_2}, & k_2 \le k_1, k_2 \le j. \end{cases}$$

4. EVALUATION

In this section we first explore the parallelism profiles of a series of DaCapo benchmarks using a single core ($k_1 = 1$). We then use these profiles to predict wall-time using more cores (e.g., $k_2 = 2$, $k_2 = 3$, and $k_2 = 4$).

4.1 Testing Environment and Settings

Experiments have been conducted on a desktop Dell Optiflex 760 PC with 8 GB of RAM equiped with an Intel Core2Quad Q9650 3.0 GHz CPU with 4 cores, referred to as Intel-4. To obtain more stable results, we disabled frequency scaling.

As software environment we use Ubuntu Linux Server 64bit version 11.04 with a custom modified version of kernel 2.6.39.4 to log thread state changes. The Java Virtual Machine (JVM) used is Oracle's Hotspot 1.6.0_23 64bit Server VM. We focus on a selection of nine DaCapo benchmarks, version 9.12 Bach (with default workload size) [3]. We based our selection on benchmarks that are CPU-bound. When necessary, we restrict the scheduling of threads to selected cores to simulate machines with a configuration different than $k = 4$ cores. To this end, we use the *taskset* command.

Since the DaCapo benchmarks have very different execution times (by default, some of them are very short), we perform a series of iterations over each test to have longer runs within the same JVM process. In more detail, a single benchmark run executes 15 iterations for *batik*, 40 for *fop*, 5 for *h2*, 5 for *jython*, 30 for *luindex*, 10 for *lusearch*, 10 for *pmd*, 10 for *sunflow*, and 10 for *xalan*. For each experiment, we measure the wall-time of the whole execution, including the JVM bootstrapping time and all the iterations. Before each measured experiment, we run the same experiment once (without measurement) to ensure that disk caches already hold the required data (e.g., the Java class files linked during benchmark execution); this helps keep the JVM startup phase (which is less interesting for us) short.

4.2 Parallelism Characterization of DaCapo Benchmarks

We used our parallellism profiler to characterize the selected DaCapo benchmarks. In this phase, we produce the input data for our prediction model. We generate a parallelism profile for each benchmark by restricting the scheduling of the whole application (including all the threads belonging to the JVM) to a single core (i.e., $k_1 = 1$).

The results are reported in Figure 1, showing the levels of parallelism reached by each benchmark and for how long during its execution time. These results represent the values observed in a single run. Several independent runs of the same benchmark show very similar execution times. The non-determinism observed during repeated runs (in separate JVM processes) is due to application-level non-determinism, differences in the thread scheduling, different identity hashcodes for objects [2], and other phenomena studied in the literature [7, 11].

The DaCapo benchmark suite is based on Java and it is an interesting evaluation environment since several system threads are automatically generated by the JVM for performing tasks such as GC, thus making the system more complex. The JVM inspects the hardware configuration to determine how many system threads to create upon startup.

Since our single-core scheduling constraint (to enforce $k_1 = 1$) is invisibile for the JVM, the total number of system threads is related to the total number of CPU cores available on the machine (Intel-4). For example, by default, the Hotspot VM used in our evaluation tends to allocate one GC thread per core.

For example, this behavior is clearly pointed out by the information gathered through our parallelism profiler for the *xalan* benchmark. According to the DaCapo documentation[2], *xalan* allocates a thread pool with a number of worker threads corresponding to the number of available CPU cores. By looking at the bars in Figure 1, *xalan* has two parallelism-level peaks at 4 and 6 threads. This means that for the majority of its execution time, *xalan* has 4 respectively 6 active threads. A comparison of the wall-time recorded for each parallelism level with the GC and just-in-time compilation times monitored through the Java ThreadMXBean API is left for future work.

The *sunflow* parallelism profile shows that the benchmark is designed to use four threads for its execution: in fact, this benchmark allocates one thread per core available to perform parallel ray-tracing. As ray-tracing is commonly known as an efficient case taking advantage from parallel execution, most of the application execution time is spent running 4 threads.

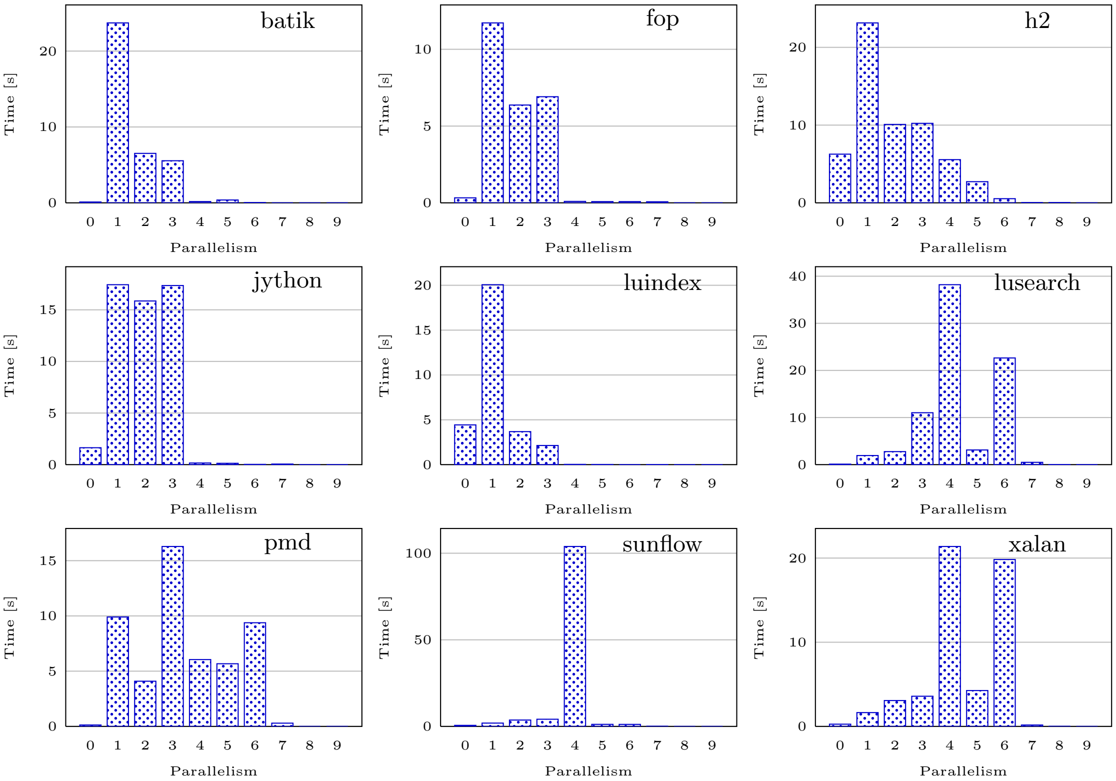
Both the *batik* and *luindex* benchmarks are mainly single-threaded (using some helper threads for a few tasks). Their parallelism profiles confirm this characteristic, showing that most of the execution time is consumed at a parallelism level of 1, while during GC phases there are several active GC threads.

The *pmd* benchmark uses one thread per hardware core to analyze a set of Java classes for a range of source code problems. Since many objects are allocated and released during the application life-time, the GC is particularly stressed. According to the documentation, this benchmark is "driven by a single client thread that is internally multi-threaded using one worker thread per hardware thread". This behavior is well visible in the parallelism profile, through the first parallelism peak reached at 1 (for the single thread client), and between 3 and 6 (when both the client and worker threads, as well as additional internal JVM threads are used).

4.3 Wall-time Prediction for DaCapo Benchmarks

Figure 2 shows the measured (░ ░) and predicted (▩ ▩) wall-time of the DaCapo benchmarks considered in our evaluation. The measurements are obtained by running the benchmarks on a varying number of cores $k_2 \in \{1, \ldots 4\}$. All predictions are based on the parallelism characterization pre-

[2] http://dacapobench.org/threads.html



par.	batik	fop	h2	jython	luindex	lusearch	pmd	sunflow	xalan
0	0.1135	0.3272	6.2622	1.6383	4.4334	0.1052	0.1286	0.6091	0.2665
1	23.7125	11.7187	23.1446	17.4342	20.0666	1.9308	9.9018	1.9215	1.6435
2	6.5175	6.3705	10.0869	15.8579	3.6819	2.7649	4.0905	3.6845	3.0751
3	5.5487	6.9116	10.2254	17.3543	2.1347	11.0191	16.2830	4.1474	3.5842
4	0.1705	0.0939	5.5579	0.1589	0.0280	38.1997	6.0497	103.8943	21.3744
5	0.3767	0.0812	2.7235	0.1283	0.0178	3.1146	5.6721	1.1907	4.2616
6	0.0333	0.0804	0.5365	0.0247	0.0001	22.6459	9.3826	1.1717	19.8270
7	-	0.0707	0.0194	0.0560	0.0106	0.4857	0.2956	0.0347	0.1675
8	-	2.40×10^{-5}	0.0260	1.40×10^{-5}	-	0.0006	0.0001	1.20×10^{-5}	0.0004
9	-	-	-	-	-	1.00×10^{-5}	-	-	1.70×10^{-5}
tot.	36.473	25.654	58.582	52.653	30.373	80.266	51.804	116.654	54.200

Figure 1: Workload parallelism characterization on Intel-4 using a single core. These measurements refer to the wall-time (in second) spent by each benchmark when a specific number of threads were active. The total time refers to the wall-time computed from the profiles.

sented in Figure 1 using the model described in Section 3. Figure 2 also presents the relative error of each prediction.

Despite of our model's simplicity, the average of the absolute relative errors is only 4.11% for $k_2 = 2$, 6.39% for $k_2 = 3$, and 6.43% for $k_2 = 4$, which can be sufficiently accurate for use in practice. Note that exact reproducibility of measurement results is typically impossible in such complex systems with many sources of non-determinism [7, 11]. Hence, also subsequent measurements usually yield different execution times.

To some extent, the prediction errors can be attributed to the perturbations introduced by our profiler. Interestingly, the wall-time prediction is particularly inaccurate for *h2* (with an absolute relative error of 15.64% for $k_2 = 4$), for which also the profiling overhead (7.98%) is four times higher than for all other benchmarks.

In general, prediction errors can be attributed to many factors that are not captured by our simple model. Our model does not consider contention (e.g., on the memory bus) that may occur when executing the application threads on a machine with more cores (i.e., $k_2 > k_1$). Furthermore, due to contention on shared data structures, compare-and-swap instructions may fail more often if more active threads execute in parallel. In addition, if active threads are spinning, executing them in parallel on additonal cores does not reduce application execution time, as it would be computed by our model.

There are several essential preconditions for the use of our prediction model. First, the application must not change its threading behavior according to the available number of CPU cores. The number of threads created by the application must be the same on the profiling machine and on the

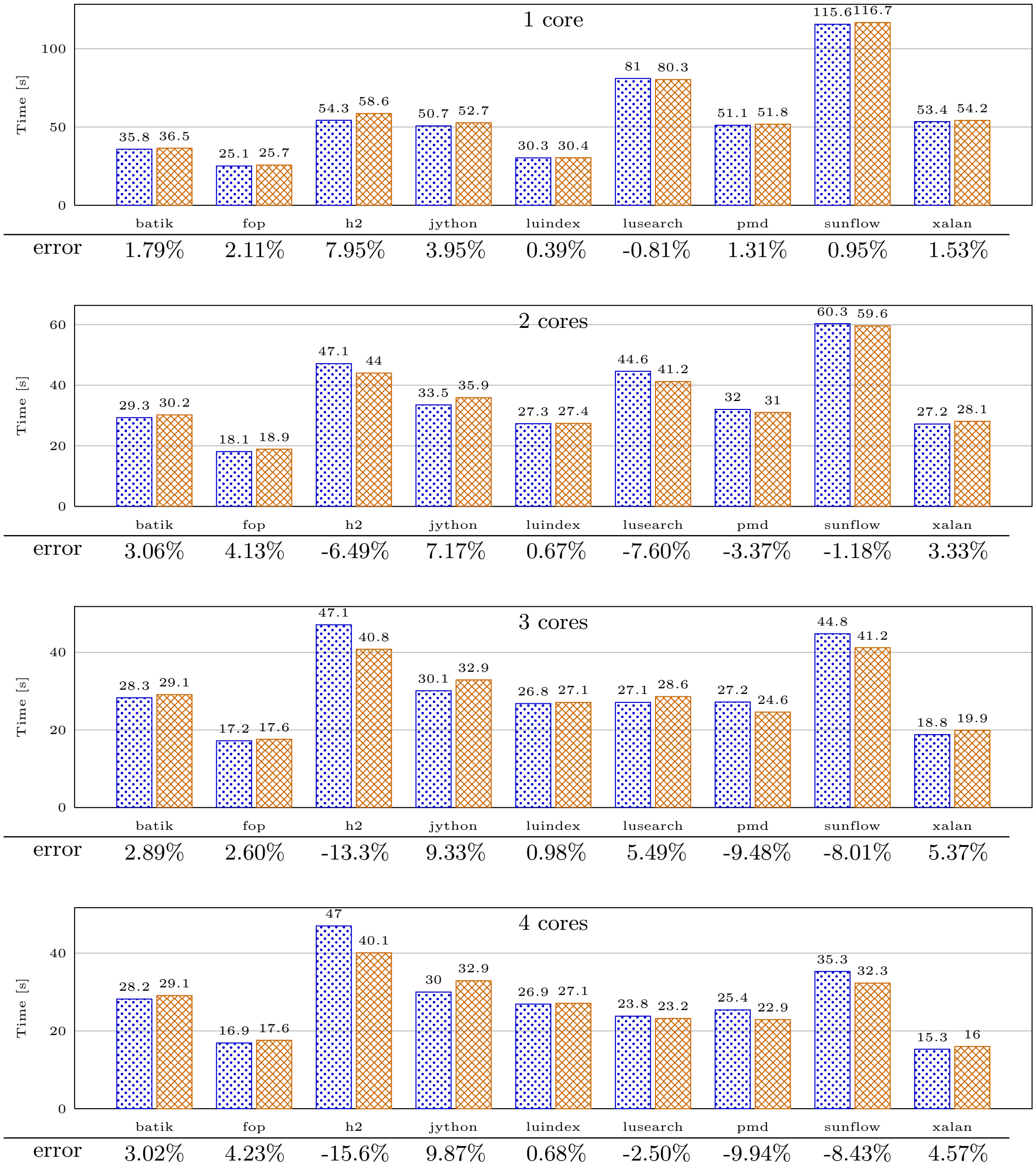


Figure 2: Performance measurement (▦) and prediction (▩) on Intel-4 running DaCapo benchmarks on 1–4 cores. Error denotes the relative error of the predicted wall-time.

target machine for prediction. In fact, both the HotSpot VM and some DaCapo benchmark adapt the number of created threads to the number of CPU cores. Hence, in our evaluation we had to ensure that Intel-4 was always seen as a quad-core machine, and we resorted to scheduler constraints to keep certain cores idle for $k_2 \in \{1, \ldots 3\}$. Second, the profiling machine and the target machine for prediction must be equivalent apart from the number of CPU cores, and the cores must operate at a constant clock speed independently of the number of active threads executed in parallel. Our measurement machine was configured to meet these constraints.

5. RELATED WORK

In this section we discuss relevant related work in the field of profiling and parallel performance prediction.

Wong et al. developed PAS2P to predict parallel application performance with an accuracy of 97% in [14]. Their approach is based on signatures created by monitoring message-passing activities, while we rely on a more generic observation of the parallelism levels and per-thread execution times.

Cook et al. used a complex statistical approach to model and predict performance of various modern architectures in [5], providing good approximations but requiring a significant effort for model calibration.

Kismet [10] is a tool that predicts parallel performance of serial (unparallelized) programs. Like our approach, Kismet does not require any modification at the program level and is based on empirical measurements, thus quickly providing results without additional efforts. Our profiler deals with multi-threaded applications, by analyizing the parallelism level of any concurrent program.

There is a significant body of research [4, 1, 6, 13, 15] addressing the performance analysis of multi-threaded applications on multicore systems, especially from the single resource's perspective, such as cache. Dey et al. [6] propose a general methodology to characterize any multi-threaded application for its last-level cache contention and private cache contention. Chen et al. [4, 1] applied queueing network models to predict application response times, relying on a profiling approach on measure resource demands, that is, required execution time on CPU and disk. Tallent and Mellor-Crummey [13] focus on Cilk programs and built a runtime performance analysis, which aims at pinpointing and quantifying serialization, using attributing work, parallel idleness, and parallel overhead to logical calling contexts. Yang et al. [15] presented a simple performance model of on-chip interconnect and intra-core bandwidth contention, for single- and multi-threaded Gaussian 03 computational chemistry code.

Our performance model is applicable to predict the overall wall-time of generic applications, instead of performance measures for single resources. Moreover, our model relies on a small set of parameters, which is readily provided by our profiling tools with moderate overheads.

Amdahl's Law [8] is widely used to predict the performance of parallel programs, ranging from the field of high performance computing to the multicore era. Hill and Marty [9] took a hardware perspective and developed a speed model for a multicore chip, applying the concept of Amdahl's Law. However, the parallelism of software aspects, that is, multi-threaded applications, is not considered.

6. CONCLUSIONS

In this short paper we presented a new profiler for characterizing the parallelism level within applications. Our profiler generates traces based on kernel scheduler events related to thread state changes. We present a prediction model using these traces that can forecast the wall-time for executing multi-threaded applications, given their parallelism profile and a target number of cores. However, the parallelism profile and the resulting wall-time predictions are platform-specific.

Using our profiler, we determined the parallelism profiles of a selection of CPU-bound DaCapo benchmarks and predicted their wall-time using a varying number of cores. Our predictions show a relative error in the range of 4.11% to 6.43% on average.

As future work, we plan to take additional factors into account, such as dynamic frequency scaling and simultaneous multi-threading to allow our model to perform predictions on hardware platforms featuring such technologies.

7. ACKNOWLEDGMENTS

The research presented in this paper has been supported by the Swiss National Science Foundation (Sinergia project CRSI22_127386 and project 200021_141002) and by the European Commission (Seventh Framework Programme grant 287746).

8. REFERENCES

[1] D. Ansaloni, L. Y. Chen, E. Smirni, and W. Binder. Model-driven consolidation of Java workloads on multicores. In *Proc. of DSN*, pages 229–234, 2012.

[2] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Softw., Pract. Exper.*, 39(1):47–79, 2009.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, Oct. 2006.

[4] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder. Achieving application-centric performance targets via consolidation on multicores: Myth or reality? In *Proc. of HPDC*, pages 229–234, 2012.

[5] J. Cook, J. Cook, and W. Alkohlani. A statistical performance model of the Opteron processor. *SIGMETRICS Perform. Eval. Rev.*, 38(4):75–80, Mar. 2011.

[6] T. Dey, W. Wang, J. Davidson, and M. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Proc. of ISPASS*, pages 76–86, 2011.

[7] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007.

[8] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Pearson Education, 2003.

[9] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE COMPUTER*, 2008.

[10] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. Kismet: parallel speedup estimates for serial programs. *SIGPLAN Not.*, 46(10):519–536, Oct. 2011.

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, Mar. 2009.

[12] A. Peternier, D. Bonetta, W. Binder, and C. Pautasso. Overseer: Low-level hardware monitoring and management for Java. In *Proc. of PPPJ*, pages 143–146, Denmark, 2011.

[13] N. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of PPoPP*, pages 229–240, 2009.

[14] A. Wong, D. Rexachs, and E. Luque. Pas2p tool, parallel application signature for performance prediction. In *Proc. of PARA*, pages 293–302, 2012.

[15] R. Yang, J. Antony, and A. P. Rendell. A simple performance model for multithreaded applications executing on non-uniform memory access computers. In *Proc. of HPCC*, pages 79–86, 2009.