Compiler Generated Progress Estimation for OpenMP Programs

Peter Zangerl, Peter Thoman, and Thomas Fahringer

University of Innsbruck, 6020 Innsbruck, Austria
{peterz,petert,tf}@dps.uibk.ac.at

Abstract. Task-parallel runtime systems have to tune several parameters and take scheduling decisions during program execution to achieve the best performance. In order to decide whether a change was beneficial to the program performance, the runtime needs some kind of feedback mechanism on the progress of the program after such a parameter change was performed. Traditionally, this feedback is derived from metrics only indirectly related to the progress of the program.
To mitigate this drawback, we propose a fully automatic compiler analysis and transformation which generates progress estimates for sequential and OpenMP programs. Combined with a runtime system interface for progress reporting this enables the runtime system to get direct feedback on the progress of the executed program.
We based our implementation on the Insieme compiler and runtime system and evaluated it on a set of eight benchmarks representing a variety of different types of algorithms. Our evaluation results show a significant improvement in estimation accuracy over traditional estimation methods, with an increasing advantage for larger degrees of parallelism.

1 Introduction

A modern runtime system needs to tune several operational parameters to better utilize the underlying hardware and achieve high performance. Examples for this kind of decisions are where to best apply dynamic voltage and frequency scaling (DVFS) [9], how to adjust the granularity of tasks or controling the amount of parallelism [5], and scheduling decisions in case a runtime system is responsible for the co-scheduling of multiple programs [13,6,10].

In order for the runtime system to measure the effectiveness of the decisions it took and reach the most effective combination of parameters, it requires some kind of feedback mechanism which provides information about the performance consequences of parameter changes – a *progress metric*. The system can then monitor the progress development and judge whether or not a particular parameter change was beneficial – enabling it to steer towards optimal settings.

In practice, there are several ways for a runtime system to estimate an application's current progress. An obvious candidate for this kind of progress information are CPU counters. A runtime system can monitor the development of certain counters and thus reason about the amount of work the application

has carried out in a given timeframe. However, there are several drawbacks to this approach: i) the CPU counters do not have a direct relationship to the application's progress; ii) counter values will also be influenced during the time spent within the runtime system itself, thus skewing the obtained results; and iii) the use of CPU counters is not portable and the desired counters might not be available on the given target hardware.

A runtime system can also take advantage of its internal state to estimate an application's progress. The runtime's task throughput is a measure of how many tasks the system finished within a given timeframe and thus is also related to the progress of the executed program. This approach has the advantage that the required values are already available in the runtime system or can be added easily without any application code modifications or special permission requirements. On the other hand, this approach is often coarse-grained and not very accurate.

Another popular alternative to the use of counters is manual instrumentation of the input code to inform the runtime system of an application's progress. This eliminates the platform dependent implementation and also is not influenced by time spent within the runtime system itself. However, this method requires a very good understanding of the input program as well as the runtime system, needs to be done manually for each program, and, due to these factors, is often either quite coarse-grained and inaccurate or labor-intensive.

To mitigate these drawbacks, we propose a novel, fully automatic compiler-based analysis and transformation to achieve accurate progress estimations in parallel applications. This enables a low-overhead and platform-independent way for parallel runtime systems to obtain direct feedback on the program's progress upon parameter changes. Our concrete contributions are as follows:

- A compiler based progress estimation analysis and transformation supporting sequential as well as parallel OpenMP input programs.
- An application programming interface for progress information collection and reporting in the runtime system.
- An implementation of the compiler analysis and runtime system facilities based on the Insieme compiler and runtime system [7].
- An evaluation of the achieved progress estimation accuracy of eight benchmark applications on a shared memory system running in different configurations, along with a comparison with the use of CPU counters, task throughput metrics and manual code instrumentation.

2 Motivation and Related Work

Any dynamic optimizing runtime system can take advantage of obtaining a *progress estimation* directly from the scheduled entities. This way, the system can evaluate the choices and parameter tuning it applied and thus steer the scheduling towards optimum settings.

Deriving an *absolute* progress completion rate towards application termination is unattainable for most non-trivial programs. Thus, one form of a good progress estimation would be a value which increases linearly and monotonously

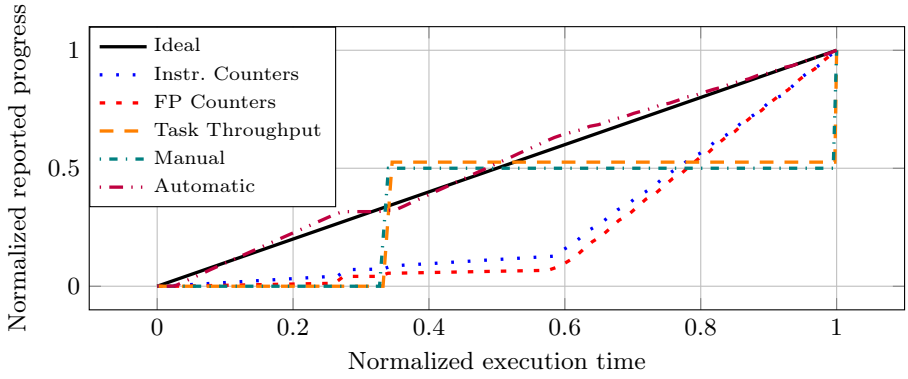

Fig. 1. Comparison of different progress reporting methods in *NPB FT*.

with the relative progress of an application. As long as the scheduled program can perform the same amount of useful work towards its goal in two observational timeframes, it should also report the same relative progress estimate.

Fig. 1 shows an illustration of different progress reporting methods during the runtime of an application by plotting the relative reported progress against the normalized time. The figure shows the results for all the progress estimation methods we evaluated in this paper for the *NPB FT* benchmark running with 64 threads (cf. Section 4). A good reporting method would report a value very close to the shown ideal line at any point during the execution. As we can see in this example, some progress estimation methods fail to achieve this criterion. This is the case for both counter approaches, which behave differently in the sequential and parallel phase of this benchmark's execution. The task throughput as well as the manual estimation approach both suffer from their coarse-grained accuracy, essentially rendering them useless for any kind of decision feedback. Our automatic approach on the other hand is able to estimate the progress quite well for most parts of the program execution.

The importance for direct feedback on the progress of scheduled applications has already been well established in the past. There is a wide body of work which tries to base scheduling decisions on the progress made by the scheduled applications to achieve optimal throughput. An example for this is the work by Wu et al. [13], where the authors introduce the concept of an application's progress based on the number of CPU cycles executed during a scheduling period. The goal of this work is a fair scheduling between equally-weighted processes where each of the applications can progress the same amount. Feliu et al. [4] follow a very similar approach. The progress of a process gets estimated by co-scheduling it in a low-contention scenario and thereby determining the maximum possible executed instructions for a given timeframe. By comparing the actual CPU counters with the maximum achievable value they determine the relative progress and use this value to create a fair co-scheduling between different processes.

The same approach has also been applied to scheduling kernels on GPUs by Anantpur et al. [1]. Lee et al. [9] additionally use counter measurements to decide when and where to best apply DVFS for reduced energy consumption while still maintaining set performance constraints. The approach presented here does not rely on CPU counters and has a more direct relation to an application's progress, enabling us to deliver more accurate and platform-independent results.

Instead of using CPU counters, Goel et al. [6] take scheduling decisions based on observing input/output events as well as inter-process communication. The approach presented by Georgakoudis et al. [5] takes into account several performance indicators and tries to build a speedup model to quantify the resulting limitations to scalability. These approaches are highly dependent on the behavior of the monitored applications, and certain programs might not generate such events for most part of their execution, resulting in unreliable estimates.

Steere et al. [10] recognize the need for a direct progress reporting mechanism between a program and its environment for improved scheduling decisions. However, they also acknowledge that it is advisable to keep these two software domains not too tightly interlocked. As a solution, they propose a *symbiotic interface* where e.g. the application notifies the operating system about data buffers and their fill-levels, enabling the latter to reason about the application's progress. The runtime interface proposed by our approach offers a way of directly reporting progress to the surrounding runtime system without burdening application developers with this task, as the invocations of this interface are created automatically with the help of a compiler component.

3 Method

Our approach combines a compiler analysis component with a task-parallel runtime system. Fig. 2 provides an overview of our proposed method. As a first step, the input program is translated into a parallelism-aware intermediate representation (IR) by the compiler frontend ①. This IR is then analyzed by our progress estimation component, which will insert reporting nodes at the appropriate locations ②. The compiler backend ③ then creates the output code to be compiled against the runtime system, resulting in the final program binary ④. The full implementation presented in this paper is publicly available[1].

3.1 Compiler Component

As we wanted our analysis to distinguish between sequential and parallel progress of an application, we decided to base it on a compiler with support for parallelism awareness in its intermediate representation. For this reason, we chose the Insieme research compiler system with its INSPIRE IR [8], as it allows us to capture the parallel semantics of a variety of input languages. While some parts of our analysis are currently tailored for OpenMP-specific semantics, it is easily extensible to other input languages supported by the Insieme compiler system.

[1] Full implementation along with instructions and evaluation script available at https://github.com/insieme/insieme/tree/progress_estimation

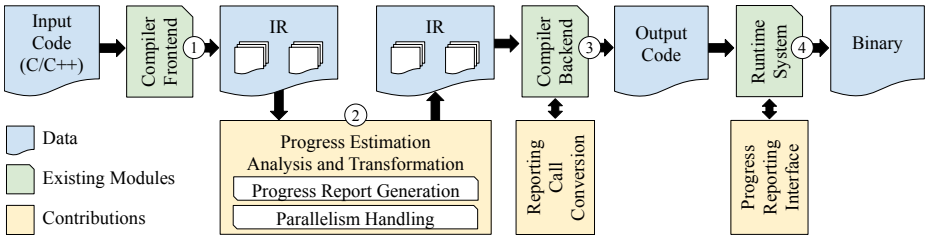

Fig. 2. Method overview for our automatic progress estimation.

Compiler Analysis The foundation for our progress estimation is a modified and extended variant of the *effort estimation* component presented by Thoman et al. [11,12]. This analysis allows us to generate effort estimations for arbitrary code parts. In this work, we define the *progress* of an application as the *accumulated effort* of its statements. In the analyses and transformations presented here, we use the same notations as Thoman et al. with the following extensions:

- *is_compound*(n) Checks whether the node n is a compound statement.
- *is_exit_point*(c, n) Checks whether node n is an exit point in compound c.
- *all_child_statements*(c) Returns all child nodes of the given compound c.
- *get_effort*(n) Returns the effort estimation for node n.
- *replace_child*(c, o, n) Replaces child o of node c with n.
- *insert_reporting_call*(c, s, p) Inserts a reporting node with progress p above node s in the compound statement c, returning zero.
- *insert_reporting_call_at_end*(c, p) Inserts a reporting node with progress p at the end of the compound statement c, returning zero.
- *conditional*$(cond, then, else)$ Refers to a conditional statement with its condition *cond* and the branch compound statements *then* and *else*.
- *loop*$(cond, body)$ Refers to a loop with its condition *cond* and the body *body*.
- *all_reporting_addresses*(n) Returns a set of all addresses rooted at node n to progress reporting nodes in any child node of n, at arbitrary depth.
- *is_openmp_{parallel/single/master}*(n) Checks whether node n represents the respective OpenMP construct in INSPIRE.
- *mark_reporting_{parallel/sequential}*(n) Replaces reporting node n with a specialized parallel or sequential version.

Progress Report Generation: A simplified version of the algorithm used to generate progress reportings is depicted by Algorithm 1. In a first phase, the analysis traverses all functions of the program. The *handle_compound* function gets passed the body, the current *progress* p and a flag r indicating whether or not to unconditionally report p at the end of the function. Each function body is analyzed statement by statement, and the effort for all the statements is evaluated (line 19). The effort of the current statement is aggregated in the current *progress estimation value* p (line 27). Before p would overflow a configurable threshold value l, we insert a new IR node into the body reporting the current value of

p, and reset p to the effort of the current statement (lines 23–25). This aggregation is applied to every statement within the compound, but several types of statements require special treatment:

- Nested compound statements are handled by recursion (lines 3–5).
- For conditional statements, we accumulate the effort for evaluating the condition (line 7) and then continue to evaluate for each branch individually, reporting at the end of each branch (lines 8–11).
- Before a loop is entered, the current value of p is always reported (line 14). Within the loop, the progress is reported before any exit-point of the loop, as well as at the end of each iteration.

Also, before each exit-point of a function, the current value of p is reported unconditionally (lines 20–21). However, in order to reduce the number of reporting instances and thus the program execution overhead, we remove instances reporting only very small values in single exit-point functions and annotate the functions with the reported value as *unreported progress*. Whenever a statement calls such a function, we then add the unreported progress to the current accumulation and thus effectively *inline* the progress reporting in this case. This optimization is not shown in Algorithm 1 for brevity.

Parallelism: This phase of the analysis is responsible for differentiating between reports in sequential and parallel code. In a second pass through the whole program we traverse all reporting nodes which have been created by the first phase. A simplified version of this transformation pass is outlined in Algorithm 2. The context of each reporting within the program is analyzed for the parallelism at its code location. This is achieved by traversing the path from each reporting location backwards up to the root of the program (line 3). We then decide on the parallel context based on what kind of OpenMP construct we meet first (lines 4 and 7). The reporting nodes are then transformed into specialized versions representing sequential or parallel progress respectively (lines 9–12). Note that, if the same function or set of functions is called in both sequential and parallel contexts, this will generate two distinct versions of these functions in the output program – this is an aspect of our automatic compiler-based system which is particularly cumbersome to replicate in a manual approach.

Tunable Parameters Our compiler component has a small set of tunable parameters influencing its behavior:

- The most important one is the *progress reporting threshold* l. This is the value above which the aggregated progress will lead to a new progress reporting node being generated within the code.
- We implemented an optimization which can be beneficial for programs which contain many very fine grained conditional statements. This optimization will – after the normal handling of conditional statements – compare the reported progress of both branches. If the reported values differ only by an

Algorithm 1 Handle Program Flow

l the progress reporting threshold

```
1: function HANDLE_COMPOUND(c, p, r)
2:     for all s ∈ all_child_statements(c) do
3:         if is_compound(s) then
4:             (s′, p) ← HANDLE_COMPOUND(s, p, ⊥)
5:             REPLACE_CHILD(c, s, s′)
6:         else if ∃cond, then, else | s = conditional(cond, then, else) then
7:             p ← p + GET_EFFORT(cond)
8:             (then′, _) ← HANDLE_COMPOUND(then, p, ⊤)
9:             REPLACE_CHILD(s, then, then′)
10:            (else′, _) ← HANDLE_COMPOUND(else, p, ⊤)
11:            REPLACE_CHILD(s, else, else′)
12:            p ← 0
13:        else if ∃cond, body | s = loop(cond, body) then
14:            p ← INSERT_REPORTING_CALL(c, s, p)
15:            eCond ← GET_EFFORT(cond)
16:            (body′, _) ← HANDLE_COMPOUND(body, eCond, ⊤)
17:            REPLACE_CHILD(s, body, body′)
18:        else
19:            p′ ← GET_EFFORT(s)
20:            if is_exit_point(c, s) then
21:                p ← INSERT_REPORTING_CALL(c, s, p + p′)
22:            else
23:                if p + p′ > l then
24:                    INSERT_REPORTING_CALL(c, s, p)
25:                    p ← p′
26:                else
27:                    p ← p + p′
28:    if r ∧ p > 0 then
29:        p ← INSERT_REPORTING_CALL_AT_END(c, p)
30:    return (c, p)
```

Algorithm 2 Handle Parallelism

m the *main* program node

```
1: for all r ∈ all_reporting_addresses(m) do
2:     par ← ⊥
3:     for all n ∈ reverse_sequence(r) do
4:         if is_openmp_parallel(n) then
5:             par ← ⊤
6:             break
7:         else if is_openmp_single(n) ∨ is_openmp_master(n) then
8:             break
9:     if par then
10:        MAKE_REPORTING_PARALLEL(r)
11:    else
12:        MAKE_REPORTING_SEQUENTIAL(r)
```

Listing 1. Runtime system API for progress reporting

```
// report sequential/global progress
void irt_report_progress(uint64_t progress);

// report parallel/per-worker progress
void irt_report_progress_thread(uint64_t progress);
```

amount less than a user-provided threshold, the reportings will be removed from the conditional branches and the analysis will continue after the conditional with the average of the removed values.
- As a last pass of the transformation, we optionally remove reportings of very small values. This is useful for programs with very intricate and tightly nested control flow, where the normal algorithm would lead to a large number of reporting nodes, each of them reporting only tiny amounts of progress.

All of these parameters can be tuned for a given use case, either to reduce the runtime overhead of our progress reporting method at the cost of slightly reduced accuracy, or alternatively to increase accuracy while potentially introducing more overhead. The default values for these parameters are set to result in reasonable compromise between low overheads and good prediction accuracy for sequential and parallel code parts alike, as shown in Section 4.

3.2 Compiler Backend

In the backend of the compiler, the reporting IR nodes need to be translated into calls which will use the runtime system's reporting facilities. The sequential and parallel version of our reporting nodes are translated into distinct runtime function calls, with the reported progress estimate being an argument of the call.

3.3 Runtime System

We extended the Insieme runtime system to support reporting of sequential as well as parallel (per-worker) progress. The runtime interface (cf. Listing 1) consists of two functions which can be used to report progress. For our prototype implementation, a periodic maintenance task within the runtime system is responsible for collecting and combining the reported progress. This thread then prints the combined application progress, allowing us to evaluate the accuracy of our approach. Additionally, these reporting facilities can also be used to implement task throughput estimation as well as manual progress reporting which we used for comparison purposes in our evaluation.

4 Evaluation

Each progress estimation method we investigated comes with a set of requirements and in return offers some features. Table 1 summarizes these properties.

Table 1. Requirements and feature set of different progress estimation methods

		CPU Counters	Task Throughput	Manual	Automatic
Requirement	Source Code Access	✗	✗	✓	✓
	Program Understanding	✗	✗	✓	✗
	Special Permissions	(✓)	✗	✗	✗
Feature	Platform Independence	✗	✓	✓	✓
	Program Independence	✓	✗	✓	✓
	Fine Granularity	✓	(✗)	(✗)	✓
	Constant Accuracy	✓	(✗)	(✗)	✓
	Low Runtime Overhead	✓	✓	(✓)	✓/✗
	Unskewed Estimate	✗	✓	✓	✓
	Per-Worker Estimate	✗	✗	(✗)	✓

Tracking an application's progress using CPU counters might require certain special permissions on some hardware platforms. More crucially, not every platform will provide all counters which we might be interested in, and different programs might be best measured by distinct counters. On the other hand, we get a very fine grained estimation with minimal overhead. However, by relying on CPU counters we work with estimates which are inherently influenced by work spent within the runtime system itself and can not get per-worker estimates.

Using the runtime's task throughput does not impose any additional requirements on the execution, as this value is readily available or easily added to an existing runtime system. However, this method does not allow per-worker performance estimates and also might work poorly with certain kinds of programs which do not produce many tasks. This also implies that its accuracy is often very fluctuating and also rather coarse-grained.

Manual and automatic compiler generated progress estimations both require the application source code in order for the necessary reporting calls to be inserted. Granularity, accuracy as well as the runtime overhead for manual estimation highly depends on how well the programmer understands the program and places the reporting calls. Most often, the result has low estimation overhead with coarse granularity and varying accuracy. Per-worker estimations are rather hard to achieve with manual progress estimation, as any code parts used in both sequential and parallel contexts have to be duplicated.

By generating the reporting calls automatically with the help of a compiler, we can mitigate most of the disadvantages of manual progress reporting, while leveraging its advantages. What remains is a certain overhead at runtime, due to the high number of reporting calls generated for high accuracy. In some programs, these overheads can be quite large and thus render a naive implementation of this approach infeasible. However, these overheads can be minimized by adjusting the tunable parameters of the compiler component (cf. Section 3.1).

Table 2. Benchmark Overview

Benchmark	Alignment	Strassen	BT	CG	EP	FT	IS	UA
Origin	AKM	Cilk	NPB					
Parameters/Class	prot.100.aa	-n 4096	B	B	B	B	C	A

4.1 Evaluation Setup

The hardware platform we are using for our evaluation is a quad-socket system with four Intel Xeon E5-4650 processors. The 8 cores (or 16 hardware threads) of each CPU are clocked at 2.7 GHz. On the software side, the system is based on CentOS 7.4 running kernel version 3.10.0-693.2.2.el7. All binaries are compiled with GCC 6.3.0 using -O2 optimizations. The thread affinity for all the executions has been fixed with a fill-socket-first policy. Each experiment has been executed ten times and we are always reporting the average values achieved.

We evaluated five different progress estimation methods in this paper, namely i) CPU counters for executed instructions; ii) CPU counters for executed floating point instructions; iii) task throughput statistics gathered in the runtime system; iv) manual progress estimation; and v) automatic compiler generated progress estimation as proposed in this paper.

4.2 Benchmarks

To evaluate the approach presented in this paper we chose a set of benchmark applications representing real-world application kernels. Table 2 lists the benchmarks used along with their origin. Most of the benchmarks originate from NASA's parallel benchmark suite [2], with the remainder being derived from the Barcelona OpenMP tasks suite [3].

4.3 Estimation Overhead

The measured overheads averaged by benchmark are shown by Fig. 3. The overhead values reported are relative to the execution of the unmodified benchmarks. Measuring the overheads did produce rather unreliable results for some benchmarks, as they showed some jitter in their execution times between successive runs at higher levels of parallelism. This is caused mainly by the non-deterministic task scheduling and work-distribution of these benchmarks.

As expected, we can observe that the overheads for both CPU counting approaches are negligible in all cases, as reading out these values during program execution should not cause significant overheads. The rather large negative overhead for the floating point counter estimate for the *UA* benchmark is a result of the execution time jitter described above, indicating that an uncertainty range of around 1% has to be considered for overhead evaluation in this benchmark.

Estimating the progress with the help of the runtime's task throughput should also not have a lot of influence on the program execution time. Still,

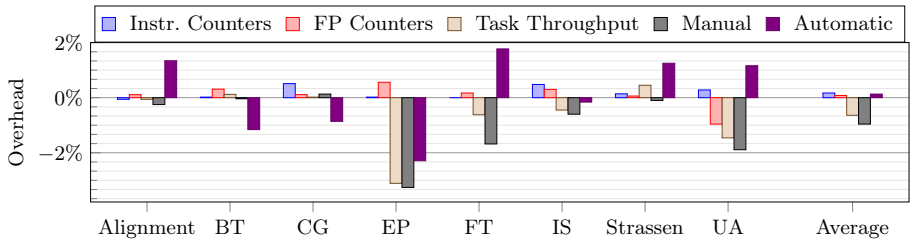

Fig. 3. Overheads for the evaluated progress estimation methods by benchmark

we can observe some small negative overheads for *FT* as well as *IS*, but especially a relatively significant negative overhead for the *EP* benchmark. Also interestingly, on average, the manual estimation method seems to actually speed up the execution of several evaluated benchmarks.

We investigated this behavior in detail, and determined that the reduction in runtime in these benchmarks is related to changes in the binary layout which occur due to the inclusion of additional functions related to progress reporting. These layout changes affect L1 instruction cache effectiveness, particularly for *EP*, and are not specific to the methods we are investigating – even adding or removing unrelated functions in the same translation units causes similar effects.

Regarding our automatic progress estimation, we can note that it shows some minor performance overhead for certain benchmarks, while it seems to improve the performance for others. The latter behavior is caused by similar effects related to the binary layout of functions in GCC as observed for the other progress metrics. Crucially, the performance overhead for our automatic progress estimation approach is less than 2% in all benchmarks.

4.4 Estimation Accuracy

For assessing the quality of the reported progress of our evaluated estimation methods we chose to employ the *mean squared error* (MSE) calculation:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{1}$$

We average the squared difference between every normalized progress report Y_i and the expected value $\hat{Y}_i$ during program execution. The latter is derived as an ideal progress estimation based on constructing a perfectly linear metric after program completion. The smaller the reported MSE, the better the estimation.

For averaging MSE values, we average over the magnitude of the error rather than the absolute value to avoid a single bad pulling the final average to non-representative high values:

$$\text{AVG}_{\text{mag}} = \frac{1}{m}\sum_{j=1}^{m} log_{10}(\text{MSE}_j) \qquad \text{AVG\_MSE} = 10^{\text{AVG}_{\text{mag}}} \tag{2}$$

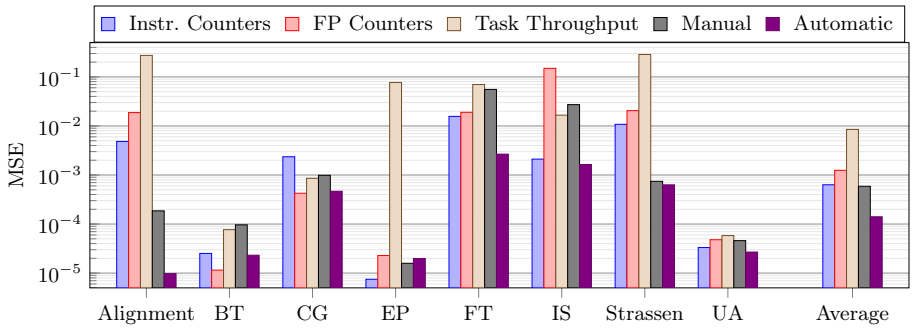

Fig. 4. Accuracy of the evaluated progress estimation methods by benchmark.

Accuracy by Benchmark Fig. 4 shows the accuracy achieved for each benchmark. All methods are able to achieve very good estimations within about the same order of magnitude for the *BT* and *UA* benchmarks. The same holds true for the *EP* benchmark, with the exception of the task throughput method. Also for the *CG* benchmark all evaluated methods result in a similar accuracy of the predictions. For the remaining four benchmarks, the achieved accuracy often diverges between the different methods by one or more orders of magnitude, with the *Alignment*-Benchmark being the most extreme example.

Accuracy by Threads The accuracy achieved by averaging our results across thread counts instead of by benchmark is shown in Fig. 5. We can observe that the estimation accuracy seems to be best for a low number of threads. The accuracy then decreases until we reach the worst results with the maximum number of threads evaluated. Moving from using all available cores to also running on all hardware threads does not have much influence on the estimation accuracy.

Regarding the specific estimation methods, we can observe that:

- The use of instruction CPU counters results in better estimates than the use of floating point CPU counters, regardless of the number of threads.
- Both counter-related estimates have a rather large drop in accuracy when moving from running on a single CPU socket to multiple sockets (16+ threads), with not too much change further on.
- The accuracy achieved by relying on the task throughput estimation is always very bad.
- Results for the manual progress estimation often fall between the accuracy achieved by the use of instruction counters and floating point counters for lower thread counts, but still are always worse than the results for our automatic estimation method.
- The automatic estimation yields the best results overall for any number of cores used, with the advantage over counter based methods increasing with higher numbers of threads.

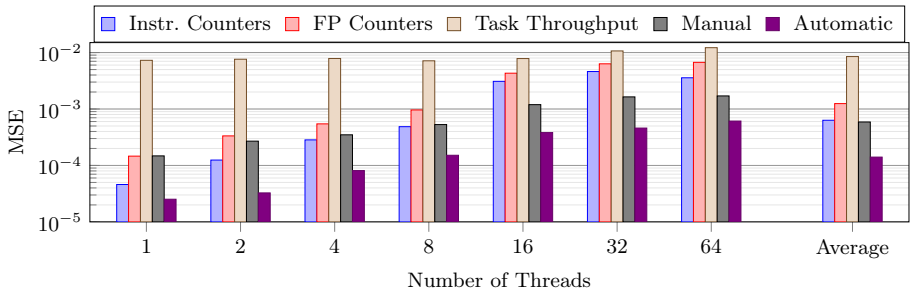

Fig. 5. Accuracy of the evaluated progress estimation methods by number of threads

The final point regarding parallelism is particularly encouraging for our approach: with hardware architectures continuously increasing in the number of cores and hardware threads per socket, it indicates that a parallelism-aware compiler-supported approach such as ours is more suitable for progress estimation on such highly parallel hardware than any of the established alternatives.

5 Conclusion

In this work, we presented a novel and fully automatic compiler analysis and transformation to generate progress estimations for OpenMP programs. Our approach provides the runtime system with direct feedback on the progress of an application, without having to resort to metrics only indirectly related to the application's progress or requiring a manual per-application implementation effort. This feedback can be used by the runtime system to measure the effectiveness of parameter changes and thus steer the execution towards optimal settings.

We evaluated our implementation on a set of eight benchmark applications implementing a wide variety of different types of algorithms. The achieved results show a good accuracy of our progress estimation, out-performing any other evaluated progress estimation method for any degree of parallelism evaluated. Crucially, the accuracy advantage of our automatic approach is increasing with a higher degree of parallelism, indicating it to be a valid approach for highly parallel future computing systems.

The work presented here offers several extension opportunities for future research. The compiler analysis itself can be further optimized to generate less reporting calls and thus runtime overhead for code parts which can be fully statically analyzed (e.g. loops with statically constant boundaries). Additionally, the set of tunable parameters of our transformation could be extended to enable a more fine-grained tradeoff between accuracy and runtime overheads. Orthogonally to the improvements of the compiler parts, future research also includes taking advantage of the generated progress estimations in the runtime system. The good accuracy of the provided estimations enables further runtime optimizations ranging from improved scheduling decisions to energy optimizations.

Acknowledgement

This work is supported by the D-A-CH project CELERITY, funded by DFG project CO1544/1-1 and FWF project 13388.

References

1. Anantpur, J., Govindarajan, R.: PRO: Progress Aware GPU Warp Scheduling Algorithm. In: 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 979–988 (May 2015)
2. Bailey, D.H., Barszcz, E., Barton, J.T., et al.: The NAS parallel benchmarks. The International Journal of Supercomputing Applications **5**(3), 63–73 (1991)
3. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: 2009 International Conference on Parallel Processing. pp. 124–131
4. Feliu, J., Sahuquillo, J., Petit, S., Duato, J.: Addressing fairness in SMT multicores with a progress-aware scheduler. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. pp. 187–196. IEEE (2015)
5. Georgakoudis, G., Vandierendonck, H., Thoman, P., Supinski, B.R.D., Fahringer, T., Nikolopoulos, D.S.: SCALO: Scalability-Aware Parallelism Orchestration for Multi-Threaded Workloads. ACM Trans. Archit. Code Optim. **14**(4), 54:1–54:25
6. Goel, A., Walpole, J., Shor, M.: Real-rate scheduling. In: Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004. pp. 434–441 (May 2004)
7. Jordan, H., Thoman, P., Durillo, J.J., Pellegrini, S., Gschwandtner, P., Fahringer, T., Moritsch, H.: A Multi-Objective Auto-Tuning Framework for Parallel Codes. In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for. pp. 1–12 (Nov 2012)
8. Jordan, H., Pellegrini, S., Thoman, P., Kofler, K., Fahringer, T.: INSPIRE: The Insieme Parallel Intermediate Representation. In: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques. pp. 7–18. PACT '13, IEEE Press, Piscataway, NJ, USA (2013)
9. Lee, S.J., Lee, H.K., Yew, P.C.: Runtime Performance Projection Model for Dynamic Power Management. In: Advances in Computer Systems Architecture. pp. 186–197. Springer Berlin Heidelberg (2007)
10. Steere, D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., Walpole, J.: A feedback-driven proportion allocator for real-rate scheduling. In: OSDI. vol. 99, pp. 145–158 (1999)
11. Thoman, P., Zangerl, P., Fahringer, T.: Task-parallel Runtime System Optimization Using Static Compiler Analysis. In: Proceedings of the Computing Frontiers Conference. pp. 201–210. ACM (2017)
12. Thoman, P., Zangerl, P., Fahringer, T.: Static Compiler Analyses for Application-specific Optimization of Task-Parallel Runtime Systems. Journal of Signal Processing Systems pp. 1–18 (2018)
13. Wu, C., Li, J., Xu, D., Yew, P.C., Li, J., Wang, Z.: FPS: A fair-progress process scheduling policy on shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems **26**(2), 444–454 (2015)