Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach*

Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer

University of Innsbruck,
Distributed and Parallel Systems Group,
A6020 Innsbruck, Austria
peter.thoman@uibk.ac.at

Abstract. The scheduling of parallel loops in OpenMP has been a research topic for over a decade. While many methods have been proposed, most focus on adapting the loop schedule purely at runtime, and without regard for the overall system state. We present a fully automatic loop scheduling policy that can adapt to both the characteristics of the input program as well as the current runtime behaviour of the system, including external load. Using state of the art polyhedral compiler analysis, we generate *effort estimation functions* that are then used by the runtime system to derive the optimal loop schedule for a given loop, work group size, iteration range and system state. We demonstrate performance improvements of up to 82% compared to default scheduling in an unloaded scenario, and up to 471% in a scenario with external load. We further show that even in the worst case, the results achieved by our automated system stay within 3% of the performance of a manually tuned strategy.

1 Introduction

OpenMP [1] is one of the most widely used languages for programming shared memory systems, particularly in the field of High Performance Computing (HPC). Despite the introduction of task-based parallelism in recent versions of the standard [5], loop parallelism remains a very important part of most OpenMP programs. Thus, the question of how to map parallel loop iterations to threads and cores has been continually investigated since the standards' inception. In Section 5 we provide an overview of some of this existing work, and describe how our approach improves upon previous methods.

Our loop scheduling system is built on the idea of *close integration between a state-of-the-art compiler providing in-depth analysis and a custom runtime library* that continuously monitors the overall system state while minimizing overhead. Such integration is realized by having the compiler generate a data

* This work was funded by the FWF Austrian Science Fund as part of project TRP 220-N23 "Automatic Portable Performance for Heterogeneous Multi-cores" and by the FFG Austrian Research Promotion Agency as part of the OpenCore project 824925.

B.M. Chapman et al. (Eds.): IWOMP 2012, LNCS 7312, pp. 88–101, 2012.

structure for each parallel loop in the original program which captures analysis-derived meta-information about the loop body in addition to the actual executable code. This approach is immediately applicable to existing programs without any code-level changes, a significant advantage considering the large number of OpenMP codes in active HPC use.

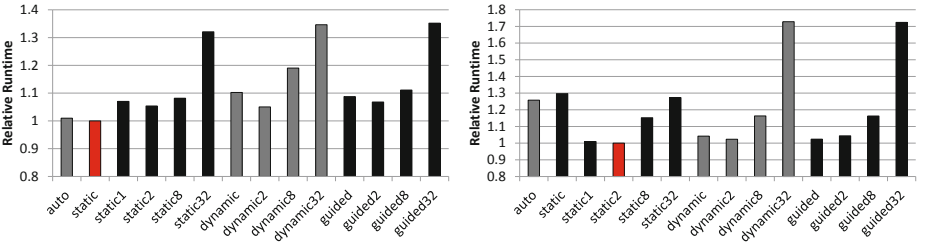
We have implemented this system and evaluated its performance. Our concrete contributions are as follows:

- A method using polyhedral model [16] based utilities to obtain effective estimates of OpenMP loop performance over all potential iteration ranges.
- A runtime scheduling algorithm that uses these estimators as well as current sytem state information to make loop scheduling decisions.
- An encoding of meta-information statically collected by the compiler into executable code usable at runtime.
- An implementation of this architecture in the Insieme compiler and runtime system [2].
- Evaluation and analysis of the actual performance of our scheduling algorithm in terms of program execution time. We compare our results to results obtained by the version of GOMP [3] included with GCC 4.5.3, using both its default scheduling policy and the best policy for each program determined by exhaustive search.

The remainder of this paper is structured as follows: The next section will provide some experimental results that motivate our work. In Section 3 we describe the architecture and implementation of our system, including the compiler analysis, the runtime scheduling system and their interaction. The results of experimental evaluation are presented in Section 4. Section 5 gathers some references to related work. Finally Section 6 presents a conclusion, and an outlook on potential future improvements.

2 Motivation

In this section we present some initial experiments using simple OpenMP kernels in a variety of settings. These results motivated our design of a unified

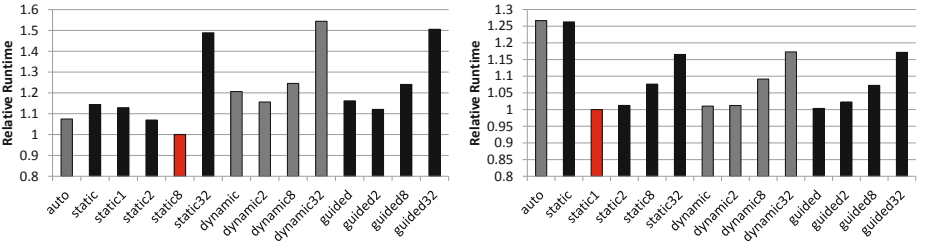


(a) Dense Matrix Multiplication (b) Triangular Matrix Multiplication

Fig. 1. Initial Experiments, Impact of Program Characteristics

compiler/runtime approach to loop scheduling. They also demonstrate the importance of load awareness. For a complete description of the experimental setup and hardware used throughout this paper see Section 4. In all our figures the relative execution time normalized to the best performing configuration is shown.

Figure 1 illustrates results for two kernels, dense matrix multiplication with full and triangular matrices, using a variety of standard OpenMP loop scheduling policies. Clearly, the ideal loop schedule depends on the characterics of the program. The dense matrix multiplication requires an equal amount of work within each iteration of the parallel loop while for the triangular matrix, the effort per iteration depends on the iterator value. We say that the dense matrix multiplication has a *flat work profile* while the work profile for the triangular matrix is *slanted*.



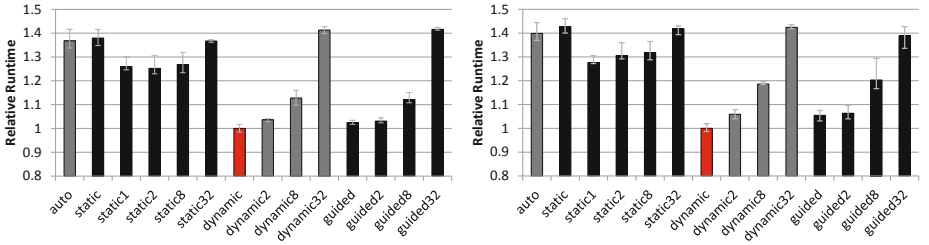
(a) Small problem size (N=160) (b) Large problem size (N=1600)

Fig. 2. Initial Experiments, Impact of Problem Size

In the next experiment we investigated the impact of the problem size on the ideal loop schedule. In Figure 2 we see that with small problem sizes, the negative performance impact of scheduling policies with a runtime component (dynamic, guided) increases, most likely due to thread scheduling overhead. Also, the increase in workload per chunk mitigates the slightly worsened load balance for a static chunk size of 8, leading to this configuration showing the best result. With large problem sizes, the relative overhead of runtime scheduling is much smaller, tough still measureable. The round-robin static scheduling policy "static,1" features acceptable load balance with relatively low overhead, making it the best performing configuration.

Finally, we look at a scenario that has often been neglected in loop scheduling research: the impact of external system load on the execution of a program. While this is an unusual situation in traditional HPC, where a cluster of servers is reserved for exclusive use by one program, it is the default on desktops, workstations and some large shared memory servers. With on-chip parallelism steadily increasing – even on embedded systems – and OpenMP being employed in end-user applications and games [6], we believe that an automatic loop scheduler needs to take this scenario into account.

Figure 3 shows the same program configurations as Figure 1(b) in two distinct load scenarios (for information on how the load simulation is performed, see Section 4). With increasing system load more fine-grained runtime scheduling



(a) Low load (desktop) scenario (b) High load (workstation) scenario

Fig. 3. Initial Experiments, Impact of External Load

policies gain a significant advantage of up to 46% compared to the default policy. These figures contain error bars since there was a slightly larger variance in the measurements – particularly for static scheduling – as a result of operating system scheduling behaviour.

To summarize, these initial findings guided the design of our loop scheduling in the following ways:

- As per the first set of figures, the automatic loop scheduler clearly needs to be aware of the *program structure*. This is accomplished via compiler analysis.
- However, as the second set of examples shows, just having static information is insufficient. The *problem size* is usually only known at runtime, necessitating integration of statuc compiler analysis with a runtime system.
- Finally, when exclusive use cannot be assumed, being aware of *external system load* is of utmost importance when selecting a scheduling policy. Thus, the runtime needs to consider the system state.

3 Architecture

Our loop scheduling system consists of the following components:

- An advanced analysis component in the Insieme source-to-source compiler that generates a symbolic *effort estimation function* for each parallel loop in the target program, or a less accurate per-iteration effort value as a fallback.
- A backend extension to the compiler that allows forwarding of this meta-information from the compiler to the Insieme runtime system.
- A monitoring component that measures the current external system load.
- A custom runtime library implementing a loop scheduling algorithm based on the meta-information provided by the compiler, the exact iteration range of the current loop and the external load.

Figure 4 illustrates how these components interact on a high level. In the following subsections each component will be discussed in detail.

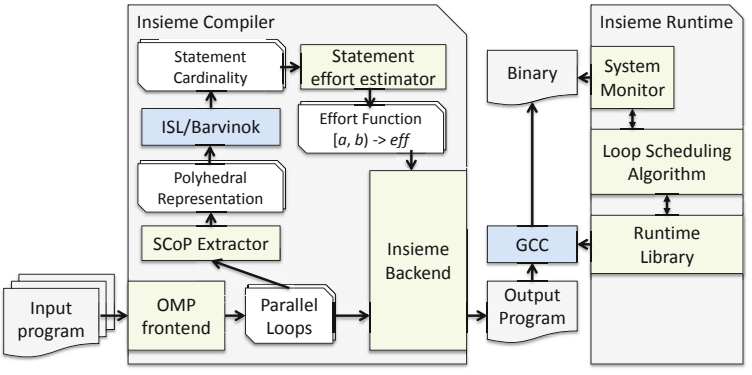


Fig. 4. An Overview of the Architecture of our System

3.1 Compiler Analysis

The main goal of our compiler analysis is to obtain, for each parallel loop, an *effort estimation function* $f_{\text{effort}} \in \mathbb{N}^2 \to \mathbb{N}$. Given lower and upper iteration bounds a and b, the evaluation of $f_{\text{effort}}(a, b)$ provides an estimate for the computational cost of the corresponding subrange of the covered loop.

This effort estimation function is derived in several steps, starting from the parallel loop body B:

1. Enclose B in a *for* loop iterating over the symbolic range $[a, b)$.
2. Extract a polyhedral representation of this parameterized loop.
3. Set the effort estimation function $f_{\text{effort}}(a, b) := 0$
4. For each statement $stmt \in B$:
   (a) Use the barvinok [17] library to obtain a piecewise affine function for the statement's cardinality $f_{\text{card}}(a, b)$
   (b) Weight this function with the effort estimation $\text{eff}(stmt)$ for the statement, computing $f_{\text{stmt}}(a, b) := f_{\text{card}}(a, b) * \text{eff}(stmt)$
   (c) Add the statement effort to the total effort function $f_{\text{effort}}(a, b) := f_{\text{effort}}(a, b) + f_{\text{stmt}}(a, b)$
5. Algebraically simplify $f_{\text{effort}}(a, b)$ using CUDD [18]

In step 2, the internal representation of the loop B is analyzed and a polyhedral representation is extracted. In-depth discussion of the polyhedral model and its application in compilers goes beyond the scope of this paper – a thorough introduction is provided by Bastoul [7]. For our purpose, it suffices to mention that the polyhedral model can be applied to Static Control Parts (SCoPs). SCoPs are program fragments that fulfill the following conditions: (1) all control structures are **for** loops or **if** statements with **affine** boundaries and conditions; (2) arrays are the only complex data structures, and are accessed with affine subscript expressions; (3) Subscripts, bounds and condition expressions depend only on loop iterators and symbolic constants.

The polyhedral model assigns to each statement an n-dimensional polytope describing how frequently it is processed within the modeled loop nest. Using

this representation, a piecewise affine function expressing the number of executions of each statement can be calculated by computing its cardinality (4a). In step 4b we arrive at an effort estimation function for each such statement by weighting its cardinality function with an estimate for the cost of executing it once. The weighting factor eff(*stmt*) takes into account the expected number of CPU instructions and memory accesses required for the given statement. This estimation is rather simplistic in our current implementation: we count the number of memory accesses and floating point operations required to perform the statement in our internal representation, without taking into account any transformations performed by the back-end compiler.

Special considerations apply when performing the SCoP analysis for our use case. Generally, the polyhedral model is used to *transform* code fragments (see section 5), while we only use it to *estimate* effort. In the former case, the analysis needs to accurately cover all effects of the code to maintain the program semantics. For estimation, failing to fully analyze some statement means that the estimation function might be less accurate, potentially weakening the performance of the scheduling algorithm, but the program semantics are preserved. In practice, this allows us to extend the applicable range of our analysis by ignoring the side effects of external function calls, as long as we can provide an effort estimate for them (e.g. `printf`). We further extended the interprocedural applicability of our estimation by applying implicit inlining which does not affect the generated code.

In the case where a loop can still not be covered by the polyhedral model, as is the case when control flow depends on input data, we apply a rough estimate to loop boundaries and conditionals to generate a single scalar effort estimation representing one iteration of the parallel loop. Section 4.2 provides some experimental data on how commonly this fallback needs to be employed in real programs.

3.2 Compiler Backend

The Insieme compiler produces C code, which is in turn translated to a binary by a secondary compiler – typically GCC. The Insieme compiler backend enumerates all the parallel loops included in the program, and, for each of them, generates a *work item structure*. To pass loop-related meta-information from the compiler to the runtime, this structure includes an (optional) function pointer of type `uint64 effort_estimator(int64 lower, int64 upper)` and a scalar fallback value `uint64 iteration_effort`. For each loop where our analysis was successful, the function pointer is set to a compiler generated C implementation of the deduced effort estimation function.

3.3 Runtime Monitoring

The resource monitoring component of the runtime needs to measure the current *external load*, that is, CPU load generated by processes other than the

managed parallel program. This is obtained by using the Linux `proc` filesystem. Specifically, the current processes' CPU usage values from `/proc/self/stat` are compared with the system-wide values obtained from `/proc/stat`, and a value between 0.0 and 1.0 representing the total external load across all cores is computed. To minimize the overhead of this method and to increase measurement reliability, this value is cached and updated at most ten times per second. Increasing the update frequency did not improve scheduling performance in our experiments.

3.4 Loop Scheduling Algorithm

All information gathered by the components outlined above is used by the runtime loop scheduler to make a scheduling decision for each individual execution of every parallel loop. The decision algorithm is outlined in Figure 5 and consists of four major steps:

1. Immediately schedule tiny loops if the estimated effort is small (lines 1-8)
2. Check the external load and use an adaptive dynamic schedule if it is greater than a threshold value (9-12)
3. If an effort estimator is available, use calculated balanced distribution (13-15)
4. Otherwise, assume irregular load and schedule dynamically (16-19)

`lower, upper`	lower and upper bound of iteration range
`members`	number of members in the current work group
`estimator`	effort estimation function for current loop
`iter_effort`	scalar per-iteration effort estimate for current loop
`load`	current external system load
`MINEFF`	minimum effort for consideration (constant per-system)
`MINLOAD`	minimum load for consideration (constant per-system)

```
1:  if estimator available then
2:      estimate = estimator(lower, upper)
3:  else
4:      estimate = (upper − lower) ∗ iter_effort
5:  end if
6:  if estimate < MINEFF then
7:      return immediate
8:  end if
9:  if load > MINLOAD then
10:     chunk = max((MINEFF/iter_effort) ∗ (1 − load), 1)
11:     return dynamic(chunk)
12: end if
13: if estimator available then
14:     shares = compute_shares(lower, upper, members, estimator)
15:     return balanced(shares)
16: else
17:     chunk = max(MINEFF/iter_effort, 1)
18:     return dynamic(chunk)
19: end if
```

Fig. 5. Loop scheduling algorithm

The result of the algorithm determines the loop scheduling behaviour for the current loop execution instance. Three modes are available:

immediate no parameters. Immediately executes the whole loop on the first thread to encounter it.

dynamic one parameter, the chunk size. Works like the standard OpenMP policy of the same name, dynamically distributing chunks of the loop range to requesting threads.

balanced requires an array of floating point values determining the relative starting points of the shares for each member of the work group. For example, [0.0, 0.25, 0.5, 0.75] would implement an equal distribution amongst four threads, while [0.0, 0.6, 0.9, 0.96] assigns progressively smaller chunks to subsequent threads.

The algorithm makes use of the compute_shares(`lower`, `upper`, `members`, `estimator`) function. It generates a distribution that tries to assign approximately the same amount of work to each member of the current work group. It first estimates the total effort for the given range [`lower`, `upper`], divides it by the number of work group members, and then uses a binary search to find a suitable chunk for each thread using the estimation function. Though this is usually a very quick process since the estimation function only takes a few cycles to run, the result is cached and reused if the same loop is executed for the same range again. This is a very common occurance in HPC codes, and the caching minimizes overhead in this case.

The parameters `MINEFF` and `MINLOAD` need to be set once per system. We have not yet developed a rigorous method for deducing these automatically. Nevertheless, experience indicates that systems are relatively insensitive regarding the precise values of these parameters, making them easy to tune manually.

4 Evaluation

In this section our system and algorithm are evaluated, starting with small kernels designed to allow easy analysis of the behaviour of the algorithm, followed by tests in a real-world setting. All experiments were performed on a SuperMicro 7046GT-TRF server with two Intel Xeon 5650 processors, containing 6 cores (12 hardware threads) each. The system runs CentOS version 5 (kernel 2.6.18) 64 bits. To compile the reference version of the example programs and as a secondary compiler for the code produced by Insieme, GCC version 4.5.3 was used with the -O3 flag set to reflect a production environment. When we refer to a "default" scheduling policy, we specifically mean the default implementation of the version of GOMP [3] included with this version of GCC.

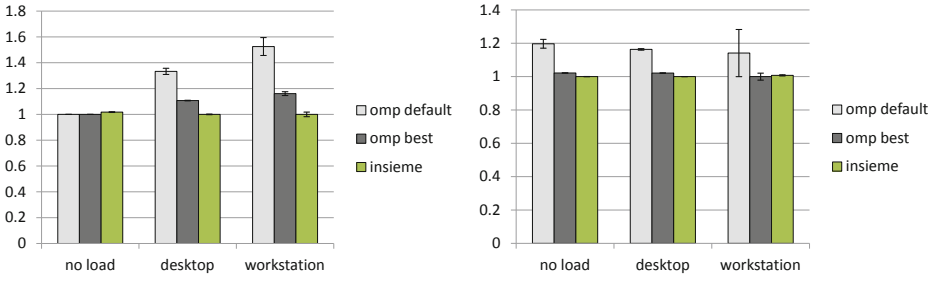
To ensure statistical significance each experiment was repeated five times, and the median result is reported. In cases where significant statistical variance occurred vertical error bars are used to show the standard deviation. We depict three values per configuration (combination of program and system load state): the default OpenMP behaviour, the best result obtained using OpenMP policies for each configuration, and the result obtained by our method. The "best" OpenMP policy is found by exhaustive search across the following settings: [(no change), auto, static, dynamic, guided]. The latter three are tested with the

chunk sizes 1, 2, 8 and 32. All values are normalized to the execution time of the best performing version.

External load profiles were recorded by monitoring each indivual core of a reference system. During experiments, these profiles were replayed by a custom load generator. We used two separate load profiles, a "desktop" profile and a "workstation" profile. The former features generally lower load and short peaks of activity, while the latter shows a higher average load level and fully saturates some cores.

4.1 Kernel Experiments

For illustrative purposes, we will apply our method to three small kernels: a dense matrix multiplication, a triangular matrix multiplication, and a pendulum simulation. These represent three major classes of problems. Both the dense and triangular matrix multiplication satisfy the SCoP constraints and can therefore be rigorously analyzed. The former has a flat work profile and is thus ideally suited to static OpenMP scheduling, while the latter has a slanted work profile. Finally, the per-iteration work in the pendulum kernel strongly depends on the input data, hence it can not be covered by SCoP analysis.



(a) Dense matrix multiplication (b) Pendulum simulation

Fig. 6. Kernel experiment

Figure 6(a) shows the results for dense matrix multiplication. In the absence of external load, fully static scheduling is ideal for this kernel, and our implementation is 1.7% slower than the best (and default) OpenMP policy. With external load, the default policy is ineffective, and our result improves on the best OpenMP policy by 10% to 15%. The best policy found for desktop load is "dynamic,8" while the best policy for the workstation load profile is "dynamic". The reason for the good result demonstrated by our method is that due to the detection of external load the chunk size is adapted dynamically.

Next, we look at the triangluar matrix multiplication kernel, which has a more interesting load profile. As Figure 7(a) illustrates, the compiler-assisted workload distribution performed by our method in the unloaded case is very effective, improving performance by 82% compared to the default behaviour,

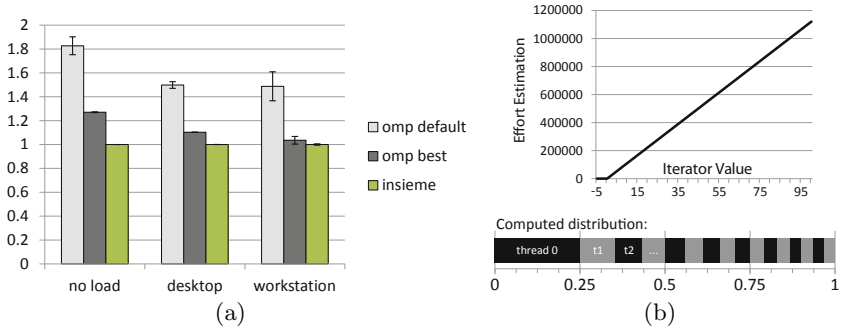


Fig. 7. Triangular matrix multiplication results

and by 27% compared to the best OpenMP scheduling policy, "static,2". This improvement over the block-cyclic scheduling can be explained by.

The effort estimation function generated by our analyis and the per-thread shares computed for 16 threads are shown in Figure 7(b). In the upper part, the effort estimation for each iterator value is plotted: iterations below zero perform no work, above that the amount of effort increases with the iterator value as the lower left triangular matrix rows become progressively wider. For this test case, the best scheduling policy with a loaded system is "dynamic" for both load profiles. Our scheduling is the fastest for both situations, though in the "workstation" case the difference is negligible (3%).

The performance results for the pendulum kernel are depicted in Figure 6(b). This benchmark computes the resting points of pendulae under the effect of magnetic fields, from many starting locations. It is communication-free but has an unpredictable, input data dependent, load imbalance, causing default scheduling to be sub-optimal. For the case with no load, the "dynamic,2" policy is best, while for the other two cases "dynamic" performs best. When the workstation external load profile is active, our method performs slightly (0.7%) worse than the "dynamic" OpenMP policy. For this load profile and the loop effort estimated for this kernel, our scheduler always decides to dynamically distribute a single loop iteration, thus performing exactly the same operation as the "dynamic" policy. The 0.7% difference can be explained by the overhead introduced by our scheduling process.

4.2 Real-World Applicability

While the results measured on small kernels are encouraging, methods based on extensive compiler analysis often fail when applied to larger code bases. However, the polyhedral model has been successfully used in production compilers [8], and, as described in Section 3.1, we were able to further relax some of its constraints for our use case.

In this section, we present an experimental analysis on some of the benchmarks contained in the NAS Parallel Benchmarks (NPB) [4] suite. As a first

step we investigate the extent to which the parallel loops contained within these programs can be treated with our analysis method.

Table 1. Applicability of our analysis on NPB loops

State	Number of loops	% of loops
Total	465	100.0%
Fully analysed	373	80.2%
Non-affine expressions	57	12.3%
Data-dependent control flow	33	7.1%
Contain while loops	2	0.4%

Table 1 lists total number of loops contained within the NPB programs, the amount that were fully analysed, and groups those that could not be analysed into categories depending on the reason for the analysis failure. Note that the number of loops listed here is higher than the amount statically contained within the program source code, due to our method analysing each call site separately. More than 4 out of 5 of all parallel loops contained in the set of benchmarks can be analyzed. The most common reason for analysis failure are non-affine boundary, condition or subscript expressions, followed by data-dependent control flow. Two of the parallel loop nests contain *while* loops.

Table 2. Nas Parallel Benchmark performance results

		Gain Over		
Name	External Load	Default	Best	Best Config
ft.B	none	4.2%	-0.2%	static,1
ft.B	desktop	21.8%	4.4%	dynamic,2
ft.B	workstation	59.9%	11.2%	dynamic
ep.B	none	14.0%	-1.9%	dynamic,8
ep.B	desktop	3.2%	-0.9%	dynamic
ep.B	workstation	19.7%	3.0%	dynamic,32
bt.B	none	-2.4%	-2.4%	static
bt.B	desktop	70.8%	65.2%	dynamic
bt.B	workstation	*	*	*
cg.B	none	8.4%	3.9%	guided,32
cg.B	desktop	113.4%	111.2%	guided,32
cg.B	workstation	471.3%	451.7%	guided,8
mg.B	none	51.7%	5.3%	dynamic
mg.B	desktop	56.1%	33.0%	dynamic
mg.B	workstation	157.4%	110.8%	dynamic,2
GM	none	13.7%	0.9%	
GM	desktop	48.2%	36.8%	
GM	workstation	94.9%	67.7%	

The results of our performance evaluation are summarized in Table 2. The "Default" and "Best" columns list the relative difference in execution time

achieved by our scheduling system compared to default scheduling (as specified by the benchmarks) and the best scheduling policy found in the search space described earlier. For example, 4.2% in the ft.B/none/default cell means that executing the ft benchmark with no external load and the default scheduling policy took 104.2% of the time the same configuration took using our scheduling system. Predefined problem size B was chosen for all the benchmarks as a good compromise between realistic size and maintaining a feasible duration for the experiments. The **GM** values are the geometric means, for each configuration, across all benchmarks.

Some points that deserve particular attention are:

- The bt benchmark with workstation external load could not be completed due to time constraints – the execution time increased disproportionately with increased load across all scheduling policies.
- There is only a single case where our algorithm performs worse than the default: bt with no load. It is the only benchmark where the default scheduling (static) is also the best policy. For most loops within bt our method picks this optimum, but for one of them the analysis fails, causing a fallback to a slightly less efficient dynamic schedule.
- The best speedup in a load-free scenario occurs for mg. This is due to the nature of the algorithm implemented by this benchmark, which leads to some loops being executed with very small iteration domains. These are identified as low-effort by our method and immediately scheduled as a whole on the first thread available.
- Generally, higher levels of external load favour our system, which can effectively adapt to them.
- Even with no external load, our method tends to achieve a marked improvement over default scheduling due to the availablity of compiler-deduced meta-information. The average speedup obtained in this setting is 13%.

5 Related Work

Enhancing OpenMP loop scheduling is a topic that has been repeatedly investigated over the years. However, most research has focused on pure runtime solutions to the problem [11][12][10]. Conversely, our approach integrates an intelligent runtime system with meta-information provided by compiler analysis.

Recent work on compiler-based OpenMP loop scheduling by Wang et al. [9] uses machine learning to estimate the best loop scheduling policy at compile time. Since this is a pure compiler approach, it cannot deal with changing runtime conditions. Also, unlike the single-pass analysis of our approach, it requires an extensive training phase.

Some systems use OpenMP in conjunction with the polyhedral model to generate parallel code [13][14]. Other recent work investigates using information provided by polyhedral analysis of OpenMP programs to improve programmer error detection [15]. None of these works aim on improving loop scheduling by forwarding static analysis results to a runtime system.

6 Conclusion

This paper presents an automatic OpenMP loop scheduling method that combines advanced compiler analysis with a load-aware runtime system. Polyhedral analysis is used to calculate a parameterized *effort estimation function* for each parallel loop, based on the cardinality of all statements it contains. Executable code for this function is generated by the compiler backend, and invoked at runtime to calculate an ideal balanced schedule or estimate efficient chunk sizes for dynamic scheduling. Additionally, external CPU load is taken into account during the scheduling process.

We evaluated our system on small kernels as well as programs from the NAS Parallel Benchmarks suite, and achieved improvements of up to 82% in the unloaded state, and 471% with heavy external load, compared to default OpenMP scheduling. To estimate the absolute effectiveness of our approach, we performed an exhaustive search over a broad range of standard OpenMP scheduling policies and compared with the best results. Our scheduling frequently improves upon even this tuned result, particularly in scenarios featuring external load. The worst-case performance achieved by our system is within 3% of the best standard OpenMP policy.

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface. Version 3.1 (July 2011)
2. The Insieme Compiler Project, `http://insieme-compiler.org/`
3. GOMP – An OpenMP implementation for GCC, `http://gcc.gnu.org/projects/gomp/`
4. Bailey, D., Barton, J., Lasinski, T., Simon, H.: The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA (1991)
5. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 100–110. Springer, Heidelberg (2008)
6. Knafla, B., Leopold, C.: Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. In: Proc. Parallel Computing (ParCo), pp. 219–226 (2007)
7. Bastoul, C.: Improving Data Locality in Static Control Programs. PhD thesis, University Paris 6, Pierre et Marie Curie, France (2004)
8. Trifunovic, K., Cohen, A., et al.: GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In: GCC Research Opportunities Workshop (GROW) (2010)
9. Wang, Z., O'Boyle, M.: Mapping parallelism to multi-cores: a machine learning based approach. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2009)
10. Zhang, Y., Burcea, M., Cheng, V., Ho, R., Voss, M.: An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In: Proc. of PDCS 2004: International Conference on Parallel and Distributed Computing Systems (2004)

11. Tzen, T., Tzen, T.H., Ni, L., Ni, L.M.: Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. IEEE Transactions on Parallel and Distributed Systems (1993)
12. Ayguadé, E., Blainey, B., Duran, A., Labarta, J., Martínez, F., Martorell, X., Silvera, R.: Is the *Schedule* Clause Really Necessary in OpenMP? In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 147–160. Springer, Heidelberg (2003)
13. Bondhugula, U., Ramanujam, J., et al.: PLuTo: A practical and fully automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI) (2008)
14. Baskaran, M., Vydyanathan, N., Bondhugula, U., Ramanujam, J., Rountev, A., Sadayappan, P.: Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2009)
15. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: `ompVerify`: Polyhedral Analysis for the OpenMP Programmer. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 37–53. Springer, Heidelberg (2011)
16. Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., Bastoul, C.: The Polyhedral Model Is More Widely Applicable Than You Think. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 283–303. Springer, Heidelberg (2010)
17. Verdoolaege, S.: barvinok: User Guide, `http://www.kotnet.org/~skimo/barvinok/barvinok.pdf`
18. Somenzi, F.: CUDD: CU Decision Diagram Package, `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`