AUTOMATICALLY TUNING PARALLEL AND PARALLELIZED PROGRAMS

Chirag Dave and Rudolf Eigenmann

Purdue University*
cdave,eigenman@purdue.edu

Abstract. In today's multicore era, parallelization of serial code is essential in order to exploit the architectures' performance potential. Parallelization, especially of legacy code, however, proves to be a challenge as manual efforts must either be directed towards algorithmic modifications or towards analysis of computationally intensive sections of code for the best possible parallel performance, both of which are difficult and time-consuming. Automatic parallelization uses sophisticated compile-time techniques in order to identify parallelism in serial programs, thus reducing the burden on the program developer. Similar sophistication is needed to improve the performance of hand-parallelized programs. A key difficulty is that optimizing compilers are generally unable to estimate the performance of an application or even a program section at compile-time, and so the task of performance improvement invariably rests with the developer. Automatic tuning uses static analysis and runtime performance metrics to determine the best possible compile-time approach for optimal application performance. This paper describes an offline tuning approach that uses a source-to-source parallelizing compiler, Cetus, and a tuning framework to tune parallel application performance. The implementation uses an existing, generic tuning algorithm called Combined Elimination to study the effect of serializing parallelizable loops based on measured whole program execution time, and provides a combination of parallel loops as an outcome that ensures to equal or improve performance of the original program. We evaluated our algorithm on a suite of hand-parallelized C benchmarks from the SPEC OMP2001 and NAS Parallel benchmarks and provide two sets of results. The first ignores hand-parallelized loops and only tunes application performance based on Cetus-parallelized loops. The second set of results considers the tuning of additional parallelism in hand-parallelized code. We show that our implementation always performs near-equal or better than serial code while tuning only Cetus-parallelized loops and equal to or better than hand-parallelized code while tuning additional parallelism.

1 INTRODUCTION

Despite much progress in automatic parallelization over the past two decades and despite the new attention this research area is receiving due to the advent

* This work was supported, in part, by the National Science Foundation under grants No. 0429535-CCF, 0751153-CNS, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

of multicore processors, parallelization is still not the default compilation option of today's compilers. One reason is that there still exist many programs in which todays parallelizing compilers cannot find significant parallelism. However, a more important reason is that, even where compilers find parallelism, they cannot guarantee that the parallelism is executed beneficially. Users may experience performance degradation unless they invest substantial time in tuning the parallel program. One of the goals behind the work presented here is to create a parallelizer that performs such tuning automatically, ensuring that compiler-parallelized code performs at least as well as the original program.

State-of-the-art solutions for finding profitable parallel regions are usually simple compile-time models. The compiler may count the number of statements and iterations of a loop and serialize the loop if these counts are known to be less than a user-defined threshold. This semi-automatic method is used in the Polaris parallelizer [1], and Intel's ICC compiler, for example. Limited compile-time knowledge of program input data, architecture parameters, and the execution environment render these models inaccurate. Several techniques have provided runtime and interactive systems for finding profitable parallel loops [2–4] and improving parallel loop performance. While these solutions have shown to be effective, they address only one of a large number of problems with compiler techniques that are limited by insufficient static knowledge. To address this issue, a secondary goal of our work is to develop a general automatic tuning solution, which can be applied to many compilation techniques. In this regard, our work is related to tuning approaches that navigate a search space of possible transformation options, picking the best through empirical evaluation [5]. This method contrasts with work that directs the search process through performance models [6]. Model-guided tuning has the potential to reduce the search space of possible compilation variants and thus tends to tune faster, while empirical tuning has the advantage of using the ultimate performance metric, which is execution time. The two methods actually complement each other and such complementary approaches have been proposed that use run-time profile information to drive optimization tuning at compile time [3, 7].

Our work also aims at a third goal. Most of today's tuning systems apply a chosen compilation variant to the *whole* program. An important un-met need is to customize compiler techniques on a section-by-section basis. Typical tuning systems choose a set of compilation flags, which are then applied to the entire program, even though some approaches have tuned individual subroutines [8]. Finer-grain tuning would allow many additional techniques and parameters to be made amenable to runtime tuning; for example, the most suitable register allocation method could be selected for each basic block. In our work, we develop a tuning method that can choose the serial or parallel execution on a loop-by-loop basis.

In this paper, we present the following contributions:

– We describe a compiler and runtime system that detects parallel loops in serial and parallel programs and selects the combination of parallel loops that shows the highest performance.

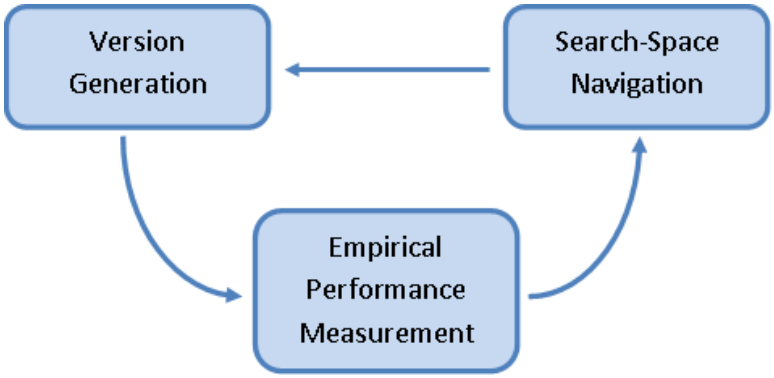

Fig. 1. High-level overview of the offline empirical tuning system. A search algorithm navigates through the space of possible optimization variants. It makes a decision on the next suitable optimization variant that the system should try. This variant is handed to the version generator in order to be compiled. Performance of each version is measured using execution time, the result of which feeds back into the search-space navigation algorithm, which determines the next suitable variant, until a convergence criterion is satisfied.

- We show that finding profitable parallelism can be done with a generic tuning method. We have chosen the *Combined Elimination Algorithm*, which was previously used to tune a generic set of compilation flags [9].
- We have developed a method that can be applied on a section by section basis, to an entire program. We use it to provide fine-grained tuning of parallelizable loops in the program.
- Using a set of NAS and SPEC OMP benchmarks, we show that our tuning method can deliver program performance that improves through automatic parallelization (of both serial and parallel program) and does not degrade below that of the original program.

The remainder of the paper is organized as follows. Section 2 presents our tuning method and its implementation using Cetus. Section 3 evaluates our techniques using available manually parallelized benchmarks. In Section 4, we draw our conclusions.

2 AUTOMATIC TUNING FRAMEWORK

This section describes the automatic tuning system. To understand the context, we briefly introduce the Cetus automatic parallelizer. The core part of our automatic tuning system is comprised of i)navigation through the search space, ii)version generation and iii)empirical evaluation of performance, as shown in Figure 1. We describe each of these in detail in the following sections.

Table 1. Number of loops automatically parallelized by Cetus (only the loop that is parallel and not enclosed by parallel loops is considered) and those already present in the input are shown in this table for a subset of the NAS Parallel and SPEC OMP2001 benchmarks. The number of Cetus-parallelized loops forms the search space for our tuning approach. Parallel loops that are nested inside outer loops can produce significant overheads due to repeatedly spawning and joining parallel threads. Selecting the right loops out of all the parallelized loops is crucial to obtaining better performance.

	BT	CG	EP	FT	LU	MG	SP	equake	art
Cetus Parallel	65	10	6	2	40	11	97	23	11
Hand Parallel	54	24	1	6	29	12	70	11	5

2.1 PROJECT CONTEXT: CETUS AUTOMATIC PARALLELIZER

The Cetus Compiler Infrastructure is a source-to-source translator for C programs [10, 11]. Input source code is automatically parallelized using advanced static analyses such as scalar and array privatization, symbolic data dependence testing, reduction recognition and induction variable substitution. The translator supports identification of loop level parallelism and OpenMP-directive based source code generation. Cetus has no compile-time heuristics or model driven approach to predict the behavior of parallelized loops and hence, it is expected that overheads associated with parallelizing small, inner loops would be significant.

Table 1 shows the number of loops parallelized by Cetus at a single level in the loop nest i.e. the outermost possible level. Some benchmarks with large, but regular loops such as BT, LU and SP from the NAS benchmark suite exhibit a larger percentage of computationally-intensive parallel loops, whereas the number drops for others i.e. for programs that exhibit loops with function calls, irregular subscripts or control flow issues as they are not amenable to auto-parallelization.

2.2 METHODOLOGY

The automatic tuner supports the automatic parallelizer in providing a parallelized version of the code that does not decrease performance below that of the original code.

This can be achieved by identifying loops that, when executed in parallel, contribute significantly to fork/join overheads or suffer from poor performance due to issues such as memory locality or load imbalance. In most cases, the size of the parallelizable loop body is too small or its iteration space is not large enough to amortize the cost of (micro)thread creation and termination. By serializing these loops during execution, the above overheads can be minimized, thus leading to more favorable execution time.

Static time models [12, 13] provide the compiler with a performance estimate based on which certain optimizations would lead to better run time. These models can be overly simplified or in many cases too complex, especially while modeling for different optimizations. They also fail to accurately capture interactions between various optimizations. Instead, our system measures whole program execution time, which is a simple, but comprehensive metric for considering *all* effects of various optimizations. This paper, however, only considers parallelization/serialization of loops in the optimization search space. No other optimizations are considered.

In order to consider the overall effects of different combinations of parallel or serial loops on the program execution time, we use an offline empirical tuning method. While we make no claims about the superiority of offline tuning versus dynamically adaptive optimization (in fact, this option was considered in prior work [14]), we have found that offline tuning performs well for most of the present benchmarks.

2.3 TUNING ALGORITHM

Empirical tuning can make use of the most powerful performance metric, which is execution time. But an exhaustive search of all the possible combinations of optimization variants can be prohibitively expensive in this case. In the context of this paper, the search space is composed of all parallelizable loops. SP has 97 parallelizable loops, and an exhaustive search would involve 2^{97} runs. A tuning algorithm logically narrows down the search space by eliminating optimization variants that may have similar effects or by deriving information related to interactions between different optimizations.

We aim at providing a generic tuning algorithm that is capable of traversing the search space of compiler optimizations, irrespective of their specific implications on the performance of the program. As described above, the overall effects of serializing/parallelizing a loop are evident in the program execution time. The Combined Elimination algorithm [9] is one such generic algorithm that has previously been used to tune a generic set of compiler flags by measuring each flag's contribution in terms of its effect on program execution time. The same algorithm is applied here to a search space composed of all the parallelizable loops in the program, each considered as a distinct on/off compiler optimization.

To the best of our knowledge, this is the first time such a tuning algorithm is being considered to study the effects of loop parallelization on a section-by-section basis for application performance. Note that the capability of the algorithm to narrow the search space by considering interactions between different compiler flags directly translates to considering interactions between parallelized or serialized combinations of loops. Such interactions make the development of a tuning method that operates on a section-by-section basis non-trivial. We do not rely on any compile-time decisions to improve our handling of these interactions and solely allow the tuning algorithm to consider them during its traversal. This, as shown in Section 3, largely improves the measurement of effects of parallelization on performance.

2.4 TUNER IMPLEMENTATION IN CETUS

Parallelization output is generated by Cetus in the form of *cetus parallel* annotations that are attached to corresponding loops in the Intermediate Representation (IR). Traversal of the IR is used to obtain this information through annotations, and build the search space of optimizations for the tuner. Also, the annotations provide a convenient way to implement fine-grained tuning on a loop-by-loop basis. This is facilitated by *selective* generation of OpenMP annotations in the output pass.

Initially, relevant information is extracted during the parallelization phase to generate the tuning search space. At every iteration of the combined elimination algorithm, the next parallelization *configuration* (a combination of optimization variants) is provided to the version generating output pass of Cetus via command line. The lexical order of appearance of a loop in the source program corresponds directly to its index in the optimization vector provided to the version generator. Different combinations of parallel loops can easily be considered for tuning by looking at Cetus-parallelized loops or loops that were OpenMP parallel in the input source itself. Figure 2 shows an example with the corresponding optimization variants that might be tested during tuning.

2.5 SYSTEM WORKFLOW

Figure 3 describes the implementation of the entire system. It is currently implemented in Python. There is no user intervention at any stage; the system is completely automated. The source program is provided to the tuning framework. The first step is auto-parallelization and identification of the number of outermost parallel loops identified by Cetus. Parallelization is performed only once and the Cetus-annotated output is stored for future reference.

A backend compiler then generates the parallel code for execution and program performance is measured at runtime. We use Intel's ICC compiler for code generation.

We first measure serial execution time and also execution time of the manually parallelized code, if available. The system measures performance using the UNIX *time* function for whole program execution; we verified that this is accurate enough for feeding back into our tuning algorithm in order to obtain the next configuration point.

The tuning process starts with a fixed initial configuration point. The optimization configuration is provided to Cetus, this time only for version generation. As shown in the example in Figure 2, OpenMP output code is generated for the current configuration of loops. Empirical evaluation returns execution time for different combinations of each base case. At every iteration for the current base case, Combined Elimination uses a decision-making criterion to provide the next configuration point. When the convergence criterion for the algorithm is met, a final trained configuration of parallel and serial loops is obtained.

```
int foo(void)
{
  #pragma cetus parallel
  #pragma cetus private(i,t)
  for ( i=0; i<50; ++i )
  {
    t = a[i];
    a[i+50] =
      t + (a[i+50]+b[i])/2.0;
  }
  #pragma cetus private(i,j,t)
  for ( i=0; i<50; ++i )
  {
    a[i] = c;
    #pragma cetus private(j,t)
    #pragma cetus parallel
    for ( j=0; j<100; j++ )
    {
      t = a[i-1];
      b[j] =
        (t*b[j])/2.0;
    }
  }
  return 0;
}
```

(a)

```
cetus -ompGen -tune-ompGen="1,1"
  (Parallelize both loops in output source)
cetus -ompGen -tune-ompGen="0,1"
cetus -ompGen -tune-ompGen="1,0"
  (Parallelize only one of the two loops)
cetus -ompGen -tune-ompGen="0,0"
  (Serialize both loops in output source)
```

(b)

Fig. 2. Parallelized source code (a) and the different variants of loop parallelization possible (b). (a) shows the output obtained from Cetus after loop parallelization. The tuner uses the number of parallel loops as its search space and then navigates this space using Combined Elimination to obtain the final binary vector configuration. Cetus' output pass is used to parse the commandline vector shown in (b) and generate OpenMP directives corresponding to the input. The entire process is automatic.

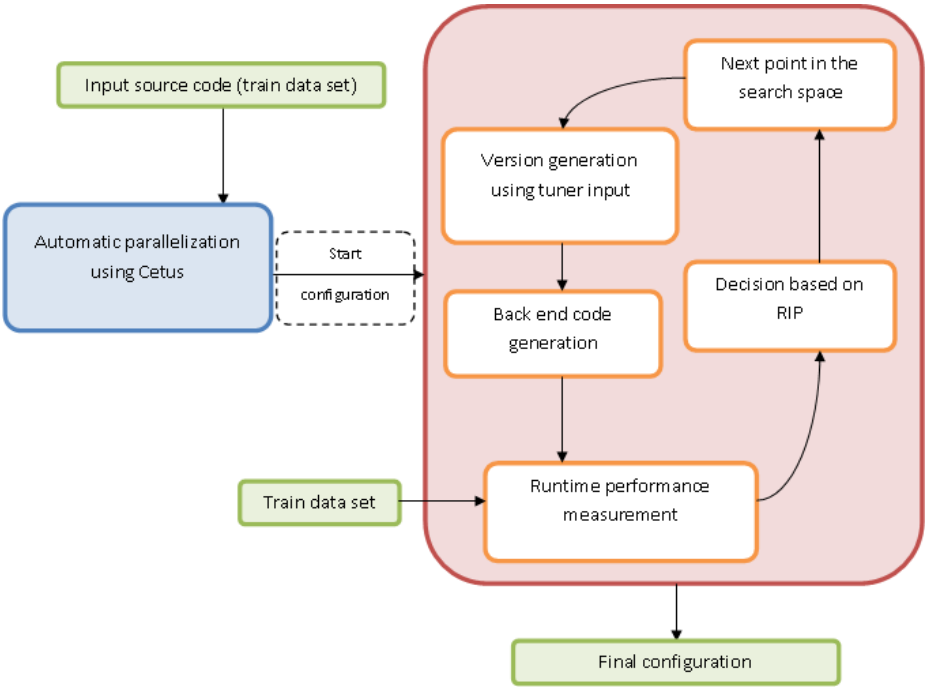

Fig. 3. This figure shows the complete training phase of our tuning implementation. Starting with auto-parallelization, the tuner starts iteration from a specific start configuration (combination of parallel loops), moving to successive configuration points on each iteration. A final configuration is derived from the trained input and is applied for evaluation to other reference data sets. (RIP: Relative Improvement Percentage)

3 EVALUATION

In this section, we evaluate the final outcome of our automatic tuning approach and compare our results specifically with the performance of the input source code. The aim is to consider the effectiveness of automatic parallelization and tuning in equaling or improving on the performance of input source code. Two tuning scenarios are considered: tuning automatically parallelized source code with no input parallelization information and tuning automatically parallelized code that is extracted in addition to the parallelism already present in the input source code. We also implemented a third scenario of using our framework to tune only hand-parallel loops. No results are presented for this case, as we didn't see significant improvements. A subset of benchmarks from the NAS Parallel benchmark suite and the SPEC OMP2001 suite was considered for evaluation.

We ran the tuning experiments in single-user mode on a Dual 2.33 GHz quad-core IntelE5410 system with 32GB memory per node. All parallel runs used the available 8 threads during execution i.e. all parallel loops executed

under the default OpenMP static scheduling model. Backend OpenMP code was generated using ICC's default optimization configuration(-O2).

We conducted the initial training of all benchmarks using a small data set. Using the training input, we obtained a configuration of loops to be parallelized/serialized. For the NAS Parallel benchmarks, the training input was the W class data set. For the SPECOMP 2001 benchmarks, we used the train data sets. Our system then applied this configuration and measured parallel performance on other data sets.

Figure 4 shows the results of training and testing tuned configurations using different input data sets. All performance results are normalized to serial execution time. *HandParallel* shows the performance of manually parallelized benchmarks. *CetusParallel* is the performance of automatically parallelized code without any tuning or performance estimation. *Cetuned* is the performance obtained using our combined Cetus and tuner approach. We only tune parallelization of Cetus-parallelized loops in this case. *Hand+Cetuned* is the performance of the benchmarks where manually parallelized code is left untouched in most cases, while additional parallelism identified by Cetus is tuned for better performance.

3.1 TUNING AUTO-PARALLELIZED CODE

This experiment represents the first scenario of automatic tuning. Only Cetus-parallelized loops were considered during parallel execution. All OpenMP directives in the input were removed. We were then dealing with programs that only consisted of loop-level parallelism. For the starting point in the tuning search space, parallelization is turned on for all parallelizable loops in the program.

Consider the training runs in Figure 4 i.e. the results obtained while training the NAS Parallel benchmarks on the W data set and the SPEC OMP2001 benchmarks on the train data set. It is clear that the *CetusParallel* version degrades performance significantly, even compared to serial execution. This can be explained by factors described in Section 2. In a large number of cases, parallelization of inner loops leads to significant fork/join overheads and hence, degradation in performance. For example, in the case of LU, Cetus identifies 40 parallel loops, 21 of which are not hand-parallelized. Some of these Cetus-parallelized loops are at the 3rd or 4th nesting level which can lead to an explosion of fork/join overheads. The same can be said for *art*, for which only 11 loops are parallelized by Cetus. Some of these are repeatedly executed inside while loops, thus increasing parallelization overheads. The Cetus-parallelized version of *art* executes for 217s as opposed to only 2.6s for the serial version, thus barely showing up on the speedup chart in Figure 4.

Even though our tuning process starts from the configuration that yields this degradation, it effectively trains itself to a configuration that nearly equals or improves over serial execution time. For example, for the above mentioned *art* result, the tuned version executes in 3s i.e. we filter out non-profitable loops to go from an execution time of 217s to 3s.

We observe a slight anomaly on some benchmarks, where the final trained configuration is poorer than serial. This could easily be avoided by setting the

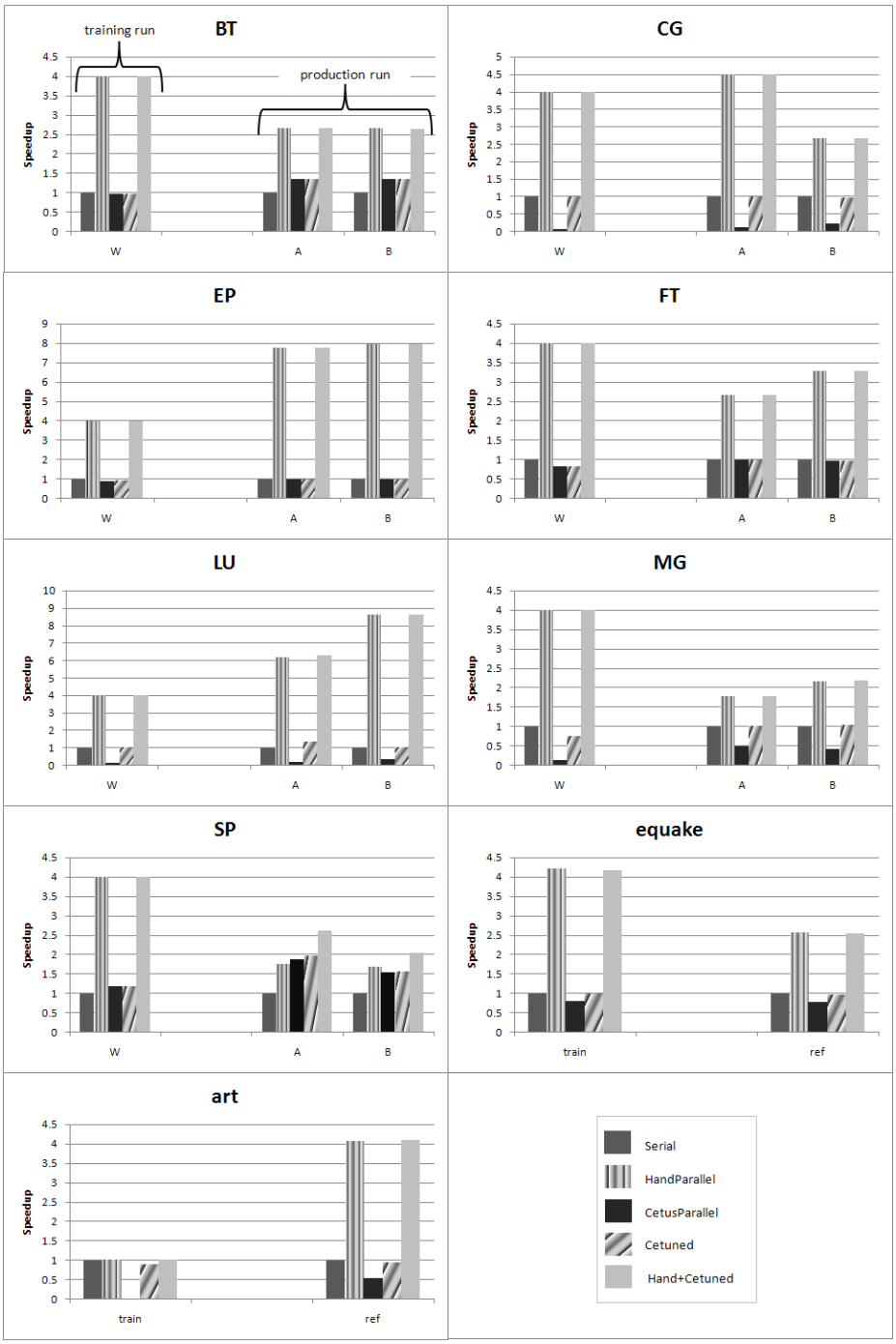

Fig. 4. Tuning results for all benchmarks. Training performed with W data set for NAS Parallel suite and *train* data set for SPEC OMP2001 suite. *CetusParallel* shows performance for compiler-parallelized code. *Cetuned* tunes only Cetus-parallelized loops and performs close to equal or better than *Serial*. *Hand+Cetuned* tunes parallel loops that exist in addition to hand-parallel loops *HandParallel*.

final configuration to serial. However, it is important to notice that the trained configuration always improves performance on the test data sets. This proves that Cetus correctly identifies parallel loops that are eventually beneficial for performance, even though it is not apparent during the training phase due to the short execution time for the train data sets. *BT* is an example of this observation.

For data classes A and B, the average improvement in execution time over serial execution was 14.5% with almost 2x increase for *SP* on data class A. Cetus parallelizes 97 loops in *SP*. Only 13 of these 97 loops are serialized in the final configuration, mostly loops with small iteration counts and small loop bodies. None of the serialized loops were hand-parallel, thus ensuring that Cetus doesn't eliminate important loops. Most importantly, 31 of the 97 loops were also hand-parallel and another 33 of the 97 loops enclosed hand-parallel loops, thus providing coarser-grain parallelism. This leads to much better performance using automatic parallelization for *SP*.

3.2 TUNING ADDITIONAL PARALLELISM IN HAND-PARALLELIZED CODE

This experiment represents the second scenario. Only Cetus-parallelized loops that were not already parallel on input or were not enclosed by loops that were parallel on input were considered in this case. In cases where Cetus identified outer-level parallel loops as opposed to those parallelized in the input, the Cetus-identified level was parallelized during evaluation. For the starting point in the tuning search space, parallelization was turned off for all additional parallelizable loops in the program.

Figure 4 shows that the training phase of tuning for additional parallel loops yields little performance benefit. Training was however performed on a small data set. The final tuned configuration did show some additional auto-parallelized loops during the training phase. These configurations were then applied to the reference data sets. In most cases, the benchmarks still behaved the same i.e. the tuning of additional parallel loops yielded little performance increase for data classes A and B. Automatic parallelization yields significant benefits in the case of *SP*, where we find parallelism at outer levels in the loop nests that are manually parallelized at inner nesting levels. Cetus parallelizes 66 additional loops, out of which 12 loops are parallelized by the final trained configuration. Out of these 12, 8 loops enclose originally hand-parallel loops and thus provide coarser-grain parallelism. *Hand+Cetuned* identifies coarser-grain parallelism as beneficial only in cases where it is beneficial. In many cases, the outer parallel loop has a very low iteration count, and hence the inner hand-parallel loop is the level defined as beneficial in the final configuration. Our *Hand+Cetuned SP* version performs almost 50% better for class A and almost 20% better for class B than manual-parallelization.

3.3 TUNING TIME

Tuning time was in the order of minutes for training each of the 9 benchmarks and eventually obtaining the best possible tuned loop configurations. Having trained on small data sets, the reference data runs would require only three runs to measure relative performance benefits of automatically parallelized and automatically tuned code, which would require time corresponding to a particular data set for the given application.

3.4 INTEL'S ICC AUTO-PARALLELIZER

We use ICC's automatic-parallelizer to study a state-of-the-art approach based on compile-time heuristics to parallelize loops. We tested the parallelizer using three different parallelizing thresholds, which can be user-defined. The auto-parallelizer uses the threshold (0-100) as a probability measure of the chances of performance improvement as determined through compile time heuristics. The default threshold of 100 means that ICC parallelizes the loop only if it is 100% sure that parallelization would yield performance benefit. This is only possible if it knows loop iteration counts at compile time. Hence, we also tested using a threshold of 99 which implies all loops considered 99% beneficial but not certainly beneficial can be parallelized. Lastly, a threshold of 0 was used, which tells ICC to parallelize all loops irrespective of performance considerations.

ICC performed extremely poorly at threshold 0 on *equake, BT, LU, MG* with upto 90% degradation in performance. The behavior at threshold 100 was inconsistent. In almost all cases, the parallelized version performed only as well as the serial execution. In the case of *LU*, the performance at threshold 100 was down 65% from serial execution. Similar results were observed at threshold 99.

The results highlight the inadequacy and inconsistency of a purely compile-time approach to automatic parallelization. Using the Cetus parallelizer and an empirical tuning approach, we obtain significantly better application performance.

3.5 DYNAMIC TUNING CONSIDERATIONS

An important observation with regards to the *art* benchmark is that the hand-parallelized version itself shows little speedup for the *train* data set but more than 4x speedup for the *ref* data set. This points to the fact that the parallelism is highly input-dependent and that, in such a case, off-line tuning using a different data set may yield little to no benefit for other data sets. Dynamic tuning would, on the other hand, be able to take input-dependent effects into consideration. However, dynamic tuning comes at the cost of incurring runtime overheads. Also, self-adapting code is unfavorable in many cases, owing to correctness and security considerations. Moreover, widely accepted techniques such as profiling for performance data are highly input dependent but are known to perform well. This argument favors our current offline tuning approach, which works well for the set of benchmarks being considered.

In the case of *art*, the *train* data set offers little opportunity to recognize beneficial parallelism. Despite this fact, the tuner identifies loops that are potentially beneficial and this is evident in the results for the *ref* data set where the tuned version improves significantly over the automatically parallelized version.

4 CONCLUSIONS

We have considered the important issue of finding profitable parallelism for auto-parallelization of both serial and parallel programs. To the best of our knowledge, we have presented the first implementation of an automatic parallelizer that nearly equals or improves in performance over the original program execution across a large set of benchmarks. Our system applies an offline tuning approach that has previously been used to tune generic compiler flags. We have adapted the algorithm to tune the parallelization of loops in the program and to apply this optimization on a loop-by-loop basis. We have shown that the programs, tuned with a training data set, always perform near-equal or better than the original code on the production data sets.

We have provided results for the implementation on a subset of the NAS Parallel and SPEC OMP2001 benchmarks. Our auto-parallelized and auto-tuned benchmarks perform 14.5% better than serial on average. Tuning additional parallelism in hand-parallelized codes performs equal to or better than the original programs (20-50% better in the case of SP).

While overzealous parallelization of loops is the biggest source of overheads in auto-parallelized programs, we intend to expand our tuning system to all optimization techniques that are available in the Cetus infrastructure. We will also consider combining empirical tuning with model-based approaches, so as to reduce tuning time as much as possible. Furthermore, for large programs, we expect that offline tuning will face limits, as different data sets may exhibit significantly different execution paths through the program. For this case, we will expand our techniques so they can be applied dynamically, during production runs of an application, without the need for a training phase.

References

1. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with Polaris. IEEE Computer **29**(12) (December 1996) 78–82
2. Voss, M.J., Eigenmann, R.: Reducing parallel overheads through dynamic serialization. In: Int'l Parallel Processing Symposium. (1999) 88–92
3. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. (2009) 177–187
4. Rauchwerger, L., Padua, D.: The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the SIGPLAN 1995 Conference on Programming Languages Design and Implementation. (jun 1995) 218–232

5. Kisuki, T., Knijnenburg, P.M., O'Boyle, M.F., Bodin, F., Wijshoff, H.A.: A feasibility study in iterative compilation. In: Proceedings of the Second International Symposium on High Performance Computing, Springer-Verlag Lecture Notes in Computer Science (1999) 121–132
6. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., Wu, P.: A comparison of empirical and model-driven optimization. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM Press (2003) 63–76
7. Wang, Z., O'Boyle, M.F.: Mapping parallelism to multi-cores: a machine learning based approach. In: Proceedings of the 14th ACM SIGPLAN Symposium on the Principles and practice of parallel programming. (2009) 75–84
8. Pan, Z., Eigenmann, R.: Fast, automatic, procedure-level performance tuning. In: Proc. of Parallel architectures and Compilation Techniques. (2006) 173–181
9. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: The 4th Annual International Symposium on Code Generation and Optimization (CGO). (March 2006) 319–330
10. : Cetus: A Source-to-Source Compiler Infrastructure for C Programs [online]. available: http://cetus.ecn.purdue.edu
11. Johnson, T.A., Lee, S.I., Fei, L., Basumallik, A., Upadhyaya, G., Eigenmann, R., Midkiff, S.P.: Experiences in using Cetus for source-to-source transformations. In: Proc. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'04), Springer Verlag, Lecture Notes in Computer Science (September 2004) 1–14
12. Baskaran, M.M., Vydyanathan, N., Bondhugula, U.K.R., 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. (2009) 219–228
13. Dou, J., Cintra, M.: Compiler estimation of load imbalance overhead in speculative parallelization. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. (2004) 203–214
14. Voss, M.J., Eigenmann, R.: High-level adaptive program optimization with adapt. In: Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'01), ACM Press (2001) 93–102