A Comparison of Online and Offline Strategies for Program Adaptation


Published in:
Proceedings of the 45th Annual ACM Southeast Conference (ACMSE)

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person’s rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.
A Comparison of Online and Offline Strategies for Program Adaptation

Matthew Curtis-Maury\(^1\), Christos D. Antonopoulos\(^2\) and Dimitrios S. Nikolopoulos\(^1\)
\(^1\)Center for High-end Computing Systems
Department of Computer Science
Virginia Tech
\{mfcurt,don\}@cs.vt.edu
\(^2\)Department of Computer Science
The College of William and Mary
and Greek Armed Forces
Division of Research and Informatics
cda@cs.wm.edu

ABSTRACT
Multithreaded programs executing on modern high-end computing systems have many potential avenues to adapt their execution to improve performance, energy consumption, or both. Program adaptation occurs anytime multiple execution modes are available to the application and one is selected based on information collected during program execution. As a result, some degree of online or offline analysis is required to come to a decision of how best to adapt and there are a variety of tradeoffs to consider when deciding which form of analysis to use, as the overheads they carry with them can vary widely in degree as well as type, as can their effectiveness.

In this paper, we attempt to qualitatively and quantitatively analyze the pros and cons of specific types of online and offline forms of information collection and analysis for use in dynamic program adaptation in the context of high performance computing. We focus on providing recommendations of which strategy to employ for users with specific requirements. To justify our recommendations we use data collected from two offline and three online analysis strategies used with a specific power-performance adaptation technique, concurrency throttling. We provide a two-level analysis, comparing online and offline strategies and then comparing strategies within each category. Our results show clear trends in the appropriateness of particular strategies depending on the length of application execution – more specifically the number of iterations in the program – as well as different expected use characteristics ranging from one execution to many, with fixed versus variable program inputs across executions.

1. INTRODUCTION
Although program adaptation is a relatively new area of research, many techniques have been proposed to allow an application to adaptively improve its execution time, energy efficiency, or both. Modern high-performance architectures contain many layers of parallelism, specifically processors, cores, and threads. These architectures leave much room for adaptation because it is not yet fully understood how best to exploit them given their inherent complexity. Further, it has been suggested that it is not possible to fully optimize an application without data collected during an execution of the program [10, 13].

Some of the more popular forms of adaptation are dynamic frequency and voltage scaling for energy, dynamically applying compiler optimizations for performance, and concurrency throttling for both. Each of these, and adaptation in general, can be performed by collecting data offline or online to make a decision on how best to adapt. Using online analysis, data is collected during test program executions to make a decision that will then be used during the live runs. Alternatively, with online analysis, the data is collected during each live execution of the application, allowing for potentially different adaptation decisions for each run. Within these two general categories exist many specific collection and analysis techniques with their own advantages and disadvantages. Given the proven effectiveness of so many forms of adaptation, a thorough study of how best to find the optimal operating point under different circumstances is both timely and necessary.

To provide insight into general patterns in the tradeoffs of different analysis strategies, we have selected specific representatives from general categories. The details of the selected techniques are presented in Section 3, however in general we picked two offline and three online strategies to evaluate. To serve as our example adaptation technique we use concurrency throttling which adapts a multithreaded application to use fewer processors, cores, and hardware threads to reduce execution time and/or energy consumption. This is an especially pertinent optimization given the growing importance of energy-efficient computing.

The results collected from our experiments show clear trends in the effectiveness and overhead of general classes of analysis strategies. From this data we are able to make recommendations about which analysis technique is most appropriate for a particular user. We feel that our results are...
representative of most adaptation strategies and so our recommendations are applicable to many users.

In the next section, we present related work and provide background. In Section 3, we describe the adaptation technique used to evaluate the online and offline analysis strategies as well as the analysis strategies themselves. Section 4 gives our experimental results and our discussion thereof, including our recommendations. Finally, we conclude in Section 5.

2. BACKGROUND

Previous work compares offline empirical-search versus offline model-based analysis within the compiler [16]. The purpose of this comparison is to determine whether adaptation of the form discussed in this paper is necessary at all. Their results show that using an accurate model for optimization of linear algebra code by ATLAS yields performance comparable to strategies employing empirical-search, with a significantly reduced offline overhead. However, the performance of the empirical-search approach is still better in most cases, and sometimes by a wide margin. This shows that program adaptation is still necessary, particularly in areas where highly accurate models do not yet exist.

There is a large body of related work on program adaptation and the work is largely split between techniques utilizing online and offline analysis strategies. In ATLAS [14], subroutines of BLAS (Basic Linear Algebra Suite) are optimized by executing each subroutine with many potential optimization parameters during offline test runs and selecting the one that performs the best for use online. Similar techniques are used in FFTW [6], SPIRAL [15], and PhiPAC [1].

Qasem et al. [11] propose an offline, profile-based analysis strategy for profitable application of loop fusion and tiling, which is difficult using compiler analysis alone as it is dependent on the underlying architecture and there are complex interactions between the two approaches. The offline analysis they perform is optimized to reduce the cost by using a realistic model to only test those configurations that are likely to perform well, rather than exhaustively searching all possible combinations. This is an effective strategy that limits the offline overhead. A similar model is used by Yotov et al. to reduce the offline search overhead of ATLAS [17].

One of the most popular forms of adaptation in the literature today is DVFS, or dynamic frequency and voltage scaling. Using this technique the processor frequency can be dynamically reduced to allow it to consume less power, while also adversely affecting performance if it can be known that its use. Even with this form of concurrency control, power and energy consumption are both generally decreased due to the use of fewer processors in many cases. The decision making process requires knowledge of the execution times of the application running on different hardware configurations, as is also the case with other adaptation approaches. Different analysis strategies, however, have different pros and cons, and the purpose of this paper is to determine under what circumstances each technique excels.

Analysis is needed to determine which hardware configuration will provide the desired properties, in terms of either performance or energy consumption. In this work, we consider adaptive concurrency throttling where the configuration with the highest expected performance is selected for use. Even with this form of concurrency control, power and energy consumption are both generally decreased due to the use of fewer processors in many cases. The decision making process requires knowledge of the execution times of the application running on different hardware configurations, as is also the case with other adaptation approaches. Different analysis strategies, however, have different pros and cons, and the purpose of this paper is to determine under what circumstances each technique excels.

In the following subsection we discuss the specific analysis techniques evaluated in this paper and how they relate to approaches presented in other work. The analysis techniques chosen are representative of larger classes of strategies, so the results are applicable to a wide range of approaches, beyond those specifically mentioned.
3.1 Offline Analysis Techniques
The first two analysis techniques that we considered perform their analysis on data collected from offline test executions of the application. This class of analysis strategies is representative of all approaches where the work is done offline, rather than during the live execution. The advantage of these approaches is that they can search for optimal operating points without contributing to the execution time of the application online, however they can be expensive in terms of their offline overhead as they must be trained separately for each application.

The first offline strategy that we tested was offline static, where the program is run once under each possible hardware configuration and the execution times are recorded. Then, the configuration with the lowest execution time is used for the live execution of the specific application. This is similar to the analysis strategy employed by Whaley, et al. [14].

The second offline strategy we used was offline phases, where the adaptation occurs at phase granularity rather than statically for the whole application execution. Program phases are simply sections where execution properties remain fairly stable. Using this technique allows for potentially better performance because different phases within an application are likely to have varying execution characteristics, resulting in different optimal configurations [12]. The decision process for this approach is similar to that of offline static except that the execution times of all offline test runs are recorded per phase, and the configuration with the lowest execution time for each phase is used online.

3.2 Online Analysis Techniques
In addition to the two offline strategies, we also selected three online analysis techniques. The first two of these work through live sampling of specific hardware configurations during runtime and the third works by making performance prediction based on a limited number of live test configurations. The online techniques exploit the iterative nature of parallel scientific applications by using the initial iterations of each phase as test executions on different configurations. All three techniques are phase aware.

3.2.1 Empirical Search
The first online strategy that we tested was online eath. Using this approach, each phase of the application is run for one iteration under each hardware configuration during the live execution. Once all configurations have been tested, the observed optimal configuration is used for each phase for the remainder of execution. This search strategy requires \( P \times C + T \) test iterations during the live execution to come to a decision, where \( P \) is the number of processors, \( C \) is the number of cores, and \( T \) is the number of hardware threads available in the system. An example from the literature of this type of analysis is presented in the STAR-MPI work of Faraj et al. [4].

We modified online eath to reduce the number of hardware configurations that are tested online, rather than exhaustively searching all configurations, since there may be many configurations on modern architectures. This strategy, called online heur, uses a hill climbing approach to first find the number of processors, then the number of cores, and finally the number of hardware threads to use for execution. At the beginning, the application is run with all execution resources, then along each dimension the resource is decremented by one until the execution time increases, executing each configuration for one iteration. This reduces the number of test configurations – and therefore iterations and online overhead – required to reach a decision to at most \( P + C + T \). Further, the configurations that are eliminated from testing are likely to be those that perform the worst, specifically those with the fewest execution resources. This strategy is similar to the heuristic work done by Frech, et al. [5] and Kistler et al. [10] to reduce the number of iterations required to find an effective configuration. Faraj et al. [4] also present an enhanced empirical search in their work on adaptively optimizing MPI collective operations.

3.2.2 Performance Prediction
To further reduce the number of iterations required to find an effective configuration, we have implemented and tested a performance prediction based approach, online pred [2]. Here the optimal configuration is not found through active sampling of different configurations. Rather, we run the application on predefined hardware configurations and use information that is collected during the execution to predict the performance on other configurations. The predictions are then used to make the decision of which configuration to use. This technique is similar to the analysis approach used by Springer, et al. [8].

On architectures with multiple layers of parallelism, only one configuration is tested to make predictions along each layer. Specifically, the application is run with all execution resources active, and from this information predictions are made for different numbers of physical processors. Then this is repeated to decide the number of cores and again for hardware threads. Under this approach, only one test run is required for each layer of parallelism in the system.

A predefined set of performance counters are collected during each test iteration. The selected counters correspond to areas of processor activity that dominate performance in multithreaded codes, such as cache misses, bus transactions, etc. During offline training, performance counter values and execution times are collected for each of the training benchmarks on each hardware configuration. We then derive coefficients using regression to calculate the instructions per cycle – a standard numeric representation of performance that is often used for performance prediction – of each hardware configuration based on the observed performance counter values during the test runs of the training benchmarks. These coefficients can be applied to counter values collected online to predict the performance of any application at runtime.

4. RESULTS AND DISCUSSION
In this section, we describe our experimental methodology as well as the results of our experiments comparing various online and offline analysis strategies for program adaptation.

4.1 Experimental Setup
In our experiments, we used the OpenMP version of the NAS Parallel Benchmark Suite 3.1 [9] (Class A) which is a collection of scientific applications parallelized using the OpenMP
The results of our performance analysis are shown in Figure 1 and the CPU energy results are shown in Figure 2. All numbers are normalized with respect to the non-adaptive strategy used with concurrency throttling on different sets of benchmarks. Lower numbers are better.

The strength of the heuristic search is its performance on applications with few iterations, where it outperforms the exhaustive search by 5%. This gain can be explained by the fact that with few iterations, the applications are more sensitive to online overhead. On the other hand, for large applications online_exh is better by 1%. The improved performance on large applications is due to their ability to amortize the increased overhead expense of a more thorough and effective search strategy.

The online_pred strategy further reduces the online search overhead by limiting the number of test executions even further, to only one per layer of parallelism on the machine. On our test machine with two layers of parallelism – processors and hardware threads – only two iterations are necessary. Using prediction based adaptation shows improvements of 18% in performance compared to naive, 10% better than online_exh and 9% better than online_heur. Further, the energy consumption is reduced by an average of 19% – better than either online_exh or online_heur. These two improvements occur because a decision is made in fewer iterations, so a larger percentage of execution is spent using configurations.
with better performance and lower energy consumption.

It is for small applications that the benefits of the reduced online overhead for *online_pred* truly appear. Compared to *naive*, prediction achieves a 25% speedup with a 29% reduction in energy consumption. Even compared to the other online analysis strategies for adaptation *online_pred* performs quite well for small applications, besting *online_cexh* by 20% and *online_heur* by 16%. For large applications, prediction is still very effective, achieving speedups of 10% compared to *naive* and 2% over the heuristic search, while matching the performance of the exhaustive search.

The results for the two offline search strategies were very similar to each other, showing a slight edge for *offline_static*. The explanation for the similarity is that while being phase-aware allows for the possibility of improved performance, *offline_phase* suffers from changing the configurations between phases, thereby hurting cache performance. However, for both application sizes, as well as overall, the two approaches see results within 0.5% of each other.

The performance of the offline analysis strategies is the best of any of the approaches tested. This is not surprising as the online overhead is reduced to zero when the work is done offline. Averaged over all benchmarks, the strategies improve performance by over 19% compared to *naive*, much better than the online empirical search strategies, but only 3% better than *online_pred*. Energy consumption is also reduced by 27% averaged over all benchmarks and 41% on benchmarks with few iterations.

The offline strategies are only slightly better than *online_pred* for small applications (by 5%), but for large applications the results are comparable between the two approaches. In fact, for large applications the choice of analysis strategy is not especially significant as they all perform within 2% of each other – approximately 10% better than *naive*. This result shows that all of the techniques are capable of finding effective operating points, however they differ in their degrees of overhead. What separates the strategies for large applications is the amount of work required offline to use each. We address this issue in more detail in following subsection.

<table>
<thead>
<tr>
<th>ONLINE</th>
<th>OFFLINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXH</td>
<td>HEUR</td>
</tr>
<tr>
<td>offline overhead</td>
<td>0</td>
</tr>
<tr>
<td>online overhead</td>
<td>P+C+T</td>
</tr>
<tr>
<td>recommendations</td>
<td>large apps; few executions</td>
</tr>
</tbody>
</table>

Table 1: The offline and online overheads of each strategy (P=#procs, C=#cores, T=#threads, L=#layers of parallelism) and recommendations of appropriate times to use each strategy, based on application size and use patterns including how many times the application will be executed.

Along with the offline overhead that must be endured by the users of each strategy. These tradeoffs suggest particular analysis strategies for users with different needs. Performance and energy results match, in general, and the recommendations for energy optimizing analysis strategies coincide with those for performance. This is true because increased online analysis increases both execution time as well as energy consumption since the selected configuration is being used for a smaller fraction of the total execution.

The *online_cexh* and *online_heur* strategies have the desirable characteristic that their offline overhead, and therefore their cost to use, is near zero. The original code must simply be modified to make calls to the adaptation library before and after each phase to allow the adaptation to occur, which could be automated by a preprocessor. For this reason, these analysis strategies are particularly appropriate for users who only intend to run an application a small number of times. While the other strategies have better results, their offline overheads make them unsuitable for an application that will only be run once – or a small number of times – because the large offline overhead will not be amortized over many executions.

Within the category of applications that are run only a small number of times, *online_heur* shows itself to be the best candidate for small applications because it outperforms *online_cexh* due to its reduced online overhead. On the other hand, the exhaustive empirical search yields better results on large applications, making it the best choice there.

For applications that will be run many times, a strategy that carries some offline overhead can be selected, because it will be made up for with the improved performance. The *offline_static* and *offline_phase* techniques are the best for applications that will be run many times, however only if the input is not changed across executions. The offline training techniques are tuned to the particular execution properties of an application and cannot adapt to them at runtime, so it is important that the offline training execution be identical to the live runs. However, changes in program input – particularly input sizes – can lead to changes in the application’s properties. As a result, offline training must be performed for each application as well as each different program input. Thus, offline strategies are limited in their applicability despite the fact that their performance is the best. They are, however, the most appropriate for precompiled libraries, such as ATLAS [14], that will be called many times. They are also appropriate for many users of scientific applications who execute the same application a large number of times.

4.3 Recommendations

Having thoroughly compared the performance and energy consumption of each analysis strategy in the previous section, we are now in a position to analyze where each approach is most appropriate. Our recommendations, shown in Table 1, are based on the performance of each approach along with the offline overhead that must be endured by the users of each strategy. These tradeoffs suggest particular analysis strategies for users with different needs. Performance and energy results match, in general, and the recommendations for energy optimizing analysis strategies coincide with those for performance. This is true because increased online analysis increases both execution time as well as energy consumption since the selected configuration is being used for a smaller fraction of the total execution.

The *online_cexh* and *online_heur* strategies have the desirable characteristic that their offline overhead, and therefore their cost to use, is near zero. The original code must simply be modified to make calls to the adaptation library before and after each phase to allow the adaptation to occur, which could be automated by a preprocessor. For this reason, these analysis strategies are particularly appropriate for users who only intend to run an application a small number of times. While the other strategies have better results, their offline overheads make them unsuitable for an application that will only be run once – or a small number of times – because the large offline overhead will not be amortized over many executions.

Within the category of applications that are run only a small number of times, *online_heur* shows itself to be the best candidate for small applications because it outperforms *online_cexh* due to its reduced online overhead. On the other hand, the exhaustive empirical search yields better results on large applications, making it the best choice there.

For applications that will be run many times, a strategy that carries some offline overhead can be selected, because it will be made up for with the improved performance. The *offline_static* and *offline_phase* techniques are the best for applications that will be run many times, however only if the input is not changed across executions. The offline training techniques are tuned to the particular execution properties of an application and cannot adapt to them at runtime, so it is important that the offline training execution be identical to the live runs. However, changes in program input – particularly input sizes – can lead to changes in the application’s properties. As a result, offline training must be performed for each application as well as each different program input. Thus, offline strategies are limited in their applicability despite the fact that their performance is the best. They are, however, the most appropriate for precompiled libraries, such as ATLAS [14], that will be called many times. They are also appropriate for many users of scientific applications who execute the same application a large number of times.
Between the two forms of offline analysis, we favor the use of phase-aware techniques like offline-phase. This is because phase-aware approaches have been generally shown to outperform static techniques [12], even if cache-effects prevent it from doing so for concurrency throttling.

Finally, online_pred has the largest offline overhead of the strategies evaluated. However, the overhead must only be paid once and prediction will work for any application of any problem size. The other online strategies also have this property, however online_pred outperforms these by a large margin on average. This advantage makes online_pred the most effective choice for applications of any size that will be run with different inputs or input sizes. Further, the training for prediction-based analysis works across applications, whereas the offline strategies require retraining for each application, so if multiple programs are to be run it is the best choice here as well. As online_pred works across applications and program inputs with a single offline training period, while still achieving comparable results to offline strategies, we feel that it is the best overall analysis strategy.

5. CONCLUSIONS

This paper has compared five different forms of analysis used for dynamic program adaptation. The specific techniques evaluated are representative of popular forms of analysis in the literature, and a thorough comparison of the strengths and weaknesses of each one was needed. The results of this study find that empirical search-based strategies are most effective when an application will be run a small number of times, due to the minimal offline overhead associated with their use. Alternatively, for applications that will be run many times with a fixed input, offline strategies are effective because they come with no online overhead. However, prediction-based analysis was shown to be the most effective approach for use with applications that will be run multiple times with different inputs or when multiple applications are going to be run with adaptation because the offline training must only be done once while its results were always within a few percent of the offline strategies.

The findings of this research are important because there has been an especially strong recent focus on finding ways of adapting applications at runtime to improve their performance or power. This interest has been due to the realization that only at runtime can full knowledge of an application become available. Our results are applicable beyond the domain of concurrency throttling and can be used both by researchers in the area of adaptation as well as users of adaptable systems.

Acknowledgments

This research is supported by the National Science Foundation (Grants CCR-0346867 and ACI-0312980) and the U.S. Department of Energy (Grant DE-FG02-06ER25751) and Virginia Tech. We would like to thank Henry P. Rose for his helpful suggestions on earlier drafts of this paper.

6. REFERENCES


