MiDataSets: Creating The Conditions For A More Realistic Evaluation of Iterative Optimization

Grigori Fursin[1], John Cavazos[2], Michael O'Boyle[2] and Olivier Temam[1]

[1] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France
{grigori.fursin,olivier.temam}@inria.fr
[2] Institute for Computing Systems Architecture, University of Edinburgh, UK
{jcavazos,mob}@inf.ed.ac.uk

Abstract. Iterative optimization has become a popular technique to obtain improvements over the default settings in a compiler for performance-critical applications, such as embedded applications. An implicit assumption, however, is that the best configuration found for any arbitrary data set will work well with other data sets that a program uses.
In this article, we evaluate that assumption based on 20 data sets per benchmark of the MiBench suite. We find that, though a majority of programs exhibit stable performance across data sets, the variability can significantly increase with many optimizations. However, for the best optimization configurations, we find that this variability is in fact small. Furthermore, we show that it is possible to find a compromise optimization configuration across data sets which is often within 5% of the best possible configuration for most data sets, and that the iterative process can converge in less than 20 iterations (for a population of 200 optimization configurations). All these conclusions have significant and positive implications for the practical utilization of iterative optimization.

1 Introduction

Iterative optimization is an increasingly popular alternative to purely static compiler optimization [17, 12, 8, 24, 7, 10, 18, 3, 20, 23]. This is due to the growing complexity of processor architectures and applications, its ability to adapt to new platforms and the fact it is a simple and systematic optimization process. However, with this approach comes new issues, e.g., the necessity to quickly explore a large optimization space [7, 10, 18, 3, 11], and the sensitivity to data sets.

Iterative optimization is based on the notion that the compiler will discover the best way to optimize a program through many evaluations of different optimization configurations. However, in reality, a user rarely executes the same data set twice. Therefore, iterative optimization is based on the implicit assumption that the best optimization configuration found will work *well* for *all data sets* of a program. To the best of our knowledge, this assumption has never been thoroughly investigated.

Most studies on iterative optimization repeatedly execute the same program/data set pair [7, 12, 10, 18, 3]. Therefore, they demonstrate the *potential* of iterative optimization, but not how it would behave in real conditions. Other studies [15] have used the `test/train/ref` data sets provided by the SPEC benchmark suite. A few studies have gone further and collected several data sets for a small number of benchmarks [5,

Questions
1. Do programs exhibit significant performance variations across data sets ?
2. More broadly, do data sets react similarly to most optimizations ?
3. Does it matter which data set you train on for iterative optimization ?
4. Can iterative optimization perform well when training is done over multiple data sets ?
5. How fast can iterative optimization converge across multiple data sets ?
6. In practice, is it possible to compare the effect of two optimizations on two data sets ?

Table 1. Key questions about the impact of data sets on iterative program optimization.

4], but these studies have only focussed on the effect of different data sets to one optimization.

In order to investigate whether a compiler can effectively discover optimizations that work well across data sets, we need to: (1) have a benchmark suite where each program comes with a significant number of data sets, (2) get a better statistical understanding of the impact of data sets on program performance and program optimization, and (3) evaluate iterative optimization under more "realistic" conditions where data sets change across executions.

We have assembled a collection of data sets, 20 per benchmark, for 26 benchmarks (560 data sets in total) of the free, commercially representative MiBench [14] benchmark suite, which especially (but not only) target embedded-like applications; we call this data set suite MiDataSets. Using this data set suite, we provide quantitative answers to 6 key questions listed in Table 1 about the impact of data sets on program performance and program optimization. Finally, we make a first attempt at emulating iterative compilation across different data sets, as it would happen in reality.

We find that, though a majority of programs exhibit stable performance across data sets, the variability can significantly increase with many optimizations. However, for the best optimization configurations, we find that this variability is in fact small. Furthermore, we show that it is possible to find a compromise configuration across data sets which is often within 5% of the best possible optimization configuration for most data sets, and that the iterative process can converge in less than 20 iterations (for a population of 200 optimization configurations).

Section 2 briefly explains how we evaluated the different data sets and varied program optimizations. Section 3 provides qualitative and quantitative answers to the questions listed in Table 1. Section 4 describes an attempt to emulate continuous optimization in a realistic setting, that is, across several data sets. Sections 5 and 6 describe related work and provide some initial conclusions.

2 Methodology

Platform, compiler and benchmarks All our experiments are performed on a cluster with 16 AMD Athlon 64 3700+ processors running at 2.4GHz, each with 64KB of L1 cache, 1MB of L2 cache, and 3GB of memory. Each machine is running Mandriva Linux 2006. To collect IPC, we use the PAPI 3.2.1 hardware counter library [1] and PAPIEx 0.99rc2 tool. We use the commercial PathScale EKOPath Compiler 2.3.1 [2] with the highest level of optimization, `-Ofast`, as the *baseline* optimization level. This compiler is specifically tuned to AMD processors, and on average performs the

Fig. 1. *Average best speedup and std. deviation across all data sets (baseline* `-Ofast` *optimization).*

same or better than the Intel 9.0 compilers on the same platform. To perform iterative optimization on each program and data set, we use a tool called PathOpt that comes with the EKOPath compiler suite. This tool can run a program with a variety of global compiler flags to find a set of flags that obtains the best performance on the targeted platform. We use 121 compiler flags that are known to influence performance and we use PathOpt to iteratively select random configurations of flags. [1] We ran the same randomly generated 200 configurations for each of the MiBench benchmarks and data sets. The PathOpt tool uses CPU execution time to compare the effects of different transformations. Figure 1 shows the average speedups achieved for each benchmark and the standard deviation across all data sets (the benchmarks are sorted by decreasing speedup).

Data sets Most of the data sets for MiBench benchmarks are based on standard file types, e.g., text, audio, image, etc. It is difficult to select a truly representative sample from the population of all possible data inputs. Instead we collected 18 different data sets per benchmark from a variety of sources including the internet, trying to vary size and nature as well as the data set properties as much as possible. For example, we collected images with various sizes, color depth, scenery, etc; texts and audio with various length and styles. Some benchmarks share the same data sets, and interestingly, the output of several benchmarks could be directly used as inputs to others. We largely decided against synthesizing data sets which would artificially exercise programs except for a few programs such as `bitcount` or `quicksort` which can readily accomodate synthetic data sets. Finally, we reused 2 data sets *small* and *large* from the original benchmarks, thus reaching 20 data sets per benchmark in total. Running all data sets usually did not require any modification of the benchmarks (except for `stringsearch` where the data set is in fact embedded in the source code). It is our intention to make these data sets publically available for other researchers to use.

[1] For more information about EKOPath compiler flags and PathOpt tool refer to the EKOPath compiler manual [2]

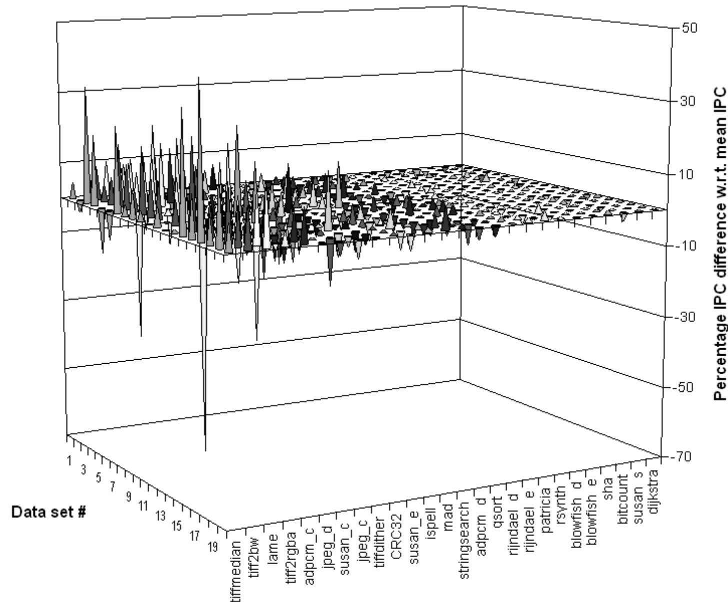

Fig. 2. *Program IPC variation across data sets (baseline* `-Ofast` *optimization).*

3 Impact of Data Sets on Program Performance and Optimization

In this section, we try to understand how varying data sets can affect program performance and iterative optimization by answering the key questions in Table 1. Beyond providing qualitative and quantitative answers to these questions, we observe a number of properties which are directly relevant to the practical implementation of iterative optimization.

3.1 Do programs exhibit significant performance variations across data sets ?

The purpose of this question is to understand whether program behavior is *stable* across data sets. One way of examining this is to observe the IPC of a program across different datasets. If the IPC varies considerably, it is likely that different parts of the program are exercised depending on data input. In such a case an optimization found by iterative compilation for a particular dataset may be inappropriate for another. It is important to note that higher IPC does *not* mean faster code. We are merely using IPC as a means of comparing the same program across different data sets,

We ran all data sets on the programs compiled with the `-Ofast` default optimization flag (our *baseline* performance). In Figure 2, we plot the percentage of IPC variation with respect to the average IPC over all data sets; the average is computed per benchmark; the benchmarks are sorted by decreasing standard deviation of IPC across all their data sets.

Figure 2 shows that MiBench benchmarks vary in their level of stability. There are very stable benchmarks across data sets such as `susan_s`, partly stable ones such as `adpcm_c` and highly varying ones such as `tiff2rgba`. `susan_s` is an application that smoothes images and has a regular algorithm with little control flow dependencies.

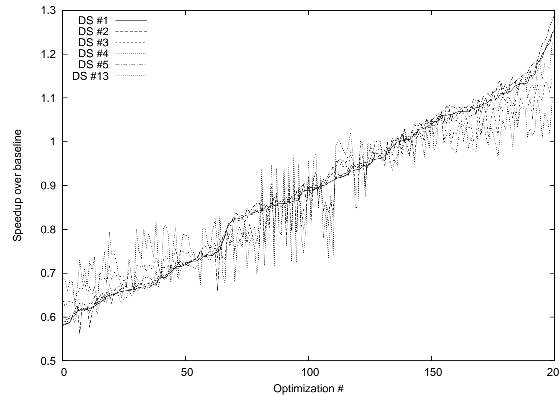

Fig. 3. *Data sets reactions to optimizations (`susan_e`).*

In this case, changing data set size or values does not influence the application's IPC. In contrast, `adpcm_c`, which is a convertor of Pulse Code Modulation (PCM) samples, and `tiff2rgba`, which is a convertor of TIFF images to RGBA color space, have most of their execution time spent in encoding/decoding subroutines. These procedures have many data and control flow dependencies and hence their behavior and IPC can vary considerably among different data sets. Still, only 6 out of 26 benchmarks can be considered unstable. The fact that the IPC of most benchmarks is fairly stable across data sets is good news for iterative optimization, as it potentially makes performance improvement across data sets possible in practice.

Yet, these experiments do not prove this trend will be consistent across all applications, or even across all data sets for a given application. For instance, `crc32` is a simple application which performs a cyclic redundancy check. The core of the program is a loop which references a large pre-defined array with a complex access function (subscript). For some function and cache size parameters, this reference could potentially result in significant conflict misses, even though `crc32` is only moderately unstable across all our data sets.

Even though there are few unstable programs, the performance variations, when they occur, are far from negligible. In 59 out of 560 experiments, a data set's IPC was more than $\pm 10\%$ away from the average, with a maximum of -64%.

3.2 Do data sets react similarly to optimizations ?

We now want to understand whether programs running different data sets react differently to program optimizations. This issue is even more closely related to iterative optimization.

Experiments show that the conclusions of Section 3.1 can in fact be misleading. Programs can have fairly consistent behavior across data sets for a given optimization configuration, e.g., the baseline `-Ofast` as in Section 3.1, but can react quite differently when optimized in a different manner. Consider benchmark `susan_e` in Figure 3; for clarity, we only plotted 6 data sets, but they are representative of the different trends over the 20 data sets. The x-axis corresponds to the optimization configuration id. As explained in Section 2, an optimization configuration is a combination of optimization

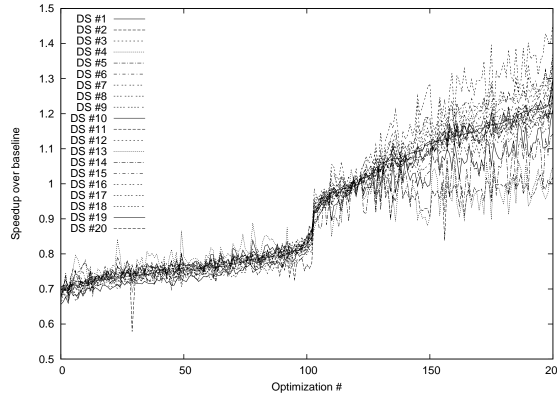

Fig. 4. *Data sets reactions to optimizations (`jpeg_d`).*

flags and parameters; recall we evaluated 200 configurations. The y-axis is the speedup over the baseline configuration. The configurations are sorted according to the speedup for data set #1; therefore, any non-monotonic behavior in one of the other data set curves means this data set reacts differently to one or several optimizations. In the rightmost 50 optimizations, the behavior of the benchmark across data sets varies wildly, with almost 30% difference for some optimizations. Similar variations are visible across the whole optimization spectrum. The behavior of `susan_e` as compared to the baseline results of Section 3.1 provides a striking contrast. When the program is compiled with `-Ofast`, its performance varies only moderately across all data sets. On the other hand, Figure 3 shows that for other optimizations this conclusion does not hold. Consequently, the results of Section 3.1 are misleading as they suggest iterative optimization can be easily applied because performance is stable. The stability properties across data sets can significantly vary with the program optimizations. As a result, iterative optimization across data sets may prove a more complicated task.

Moreover, other programs can naturally exhibit strong variations for many optimizations. Consider, for instance, `jpeg_d` in Figure 4; all 20 data sets are plotted. For some optimizations, the speedup difference across data sets can exceed 50%. Unfortunately, even though the rightmost 50 optimizations are consistently among the top performers across all data sets and benchmarks, the higher their potential speedup, the higher their variability as well.

Still, for some programs, the speedup variations across all optimizations can be strong, but very consistent across data sets, and thus confirm the stability observed in Section 3.1. Consider `dijkstra` in Figure 5; all 20 data sets have been plotted. Performance is obviously very stable across all data sets and optimizations, even though the speedups are far from negligible.

Overall, these three programs are fairly representative of the different possible behavior of the remaining benchmarks, and we roughly found the same number of benchmarks belong to each of the three aforementioned categories. Consequently, a significant number of benchmarks are not so stable across data sets when factoring in the optimizations, thereby complicating the iterative optimization task.

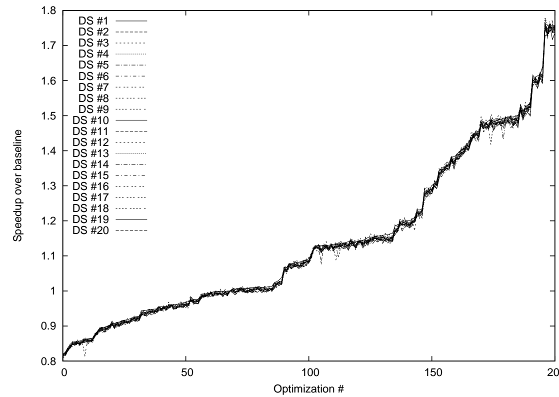

Fig. 5. *Data sets reactions to optimizations (`dijkstra`).*

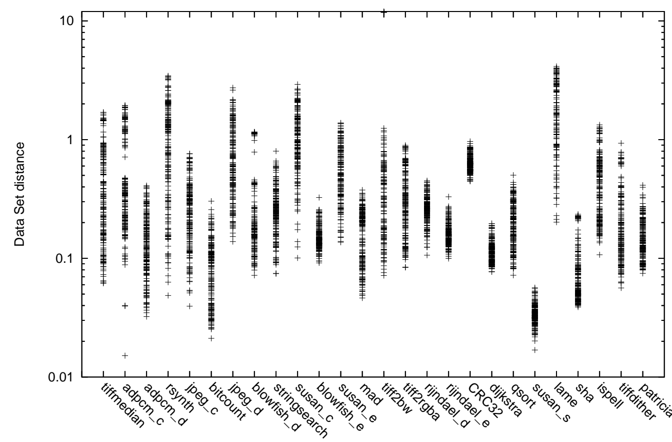

Fig. 6. Data set distances.

In order to summarize the stability of programs across data sets, we introduce a *data set distance* metric. We define the distance d between two data sets i, j as: $d = \sqrt{\sum_{1 \leq opt \leq 200} (s^i_{opt} - s^j_{opt})^2}$, where *opt* denotes an optimization configuration. This distance characterizes how differently two data sets react to optimizations. The repartition of distances of all possible pairs of data sets for a program is itself an indication of the program stability across data sets. On purpose, the metric is based on the *absolute* speedup difference rather than the relative speedup, in order to compare distances across programs. We compute the distances between all pairs of data sets for each program, and report them in Figure 6. Programs like `dijkstra`, which are very stable across data sets have small distances, no more than 0.1 to 0.5, while unstable programs like `jpged_d` have significant distances, more than 1; the maximum reported distance is greater than 10 (see `tiff2bw`). At least 8 programs have rather large data set distances, characteristic of their unstable behavior across optimizations.

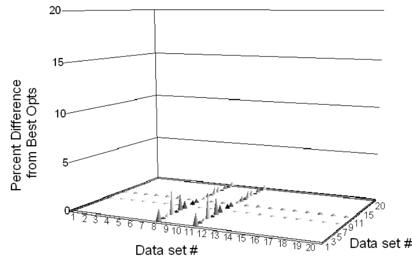

Fig. 7. Best optimizations across data sets (`sha`)

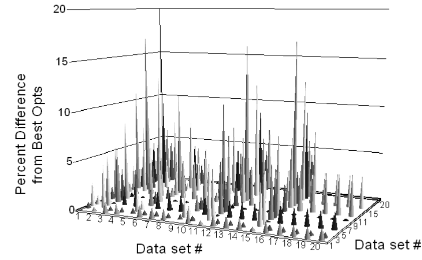

Fig. 8. Best optimizations across data sets (`susan_c`)

3.3 Does it matter which data set you train on for iterative optimization ?

Because iterative optimization techniques almost systematically experiment with single data sets, they both *train* and *test* on the same data set. With more data sets, it is possible to move closer to a more realistic implementation of iterative optimization where train and test sets are distinct.

In order to evaluate the impact of this more realistic situation, as well as the choice of the training data set, we implement a simple iterative optimization strategy where we train on a single data set and test on the 19 remaining data sets. For the training, we exhaustively search the 200 optimizations on the single data set. We then select the best configuration C for this data set, and run the remaining 19 data sets on the program optimized with C. We then measure the difference between the performance obtained with C and with the actual best possible optimization for each data set as shown in Figures 7 and 8, where the x-axis corresponds to the training data set, the z-axis corresponds to the testing data set.

While Section 3.2 shows that a significant fraction of the programs can react fairly differently across data sets for many optimizations, it turns out that, for the best optimization, programs have little or no variability with respect to data sets. Most programs show similar trends to Figure 7. Out of all the 26 benchmarks that we studied, 20 benchmarks show little variance when using the best optimization configuration of the different data sets, therefore any data set could be used for iterative optimization. The best optimization for that program will work well regardless of the data set used. This has positive implications for the practical application of iterative optimization. However, it should also be remebered that the sample number of configurations considered (200) is small compared to the entire optimization space. It may be the case that for other non-sampled configurations, that variation is much greater.

However there is a small group of programs that show medium to high variability with regards to the best optimization among data sets. For this group of programs, a user cannot simply use any single data set for iterative optimization if they wish to obtain the best performance for the rest of that program's data sets. Using the best optimization configuration of iterative optimization for any particular data set can significantly degrade performance when used on other data sets of the same program over the com-

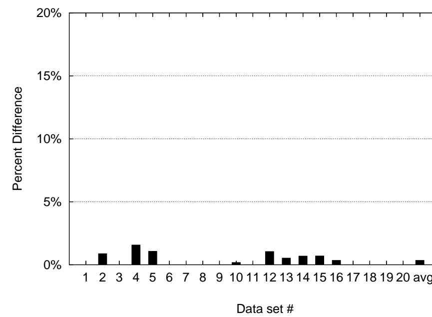

Fig. 9. *Average best opt relative to best opt(`dijkstra`).*

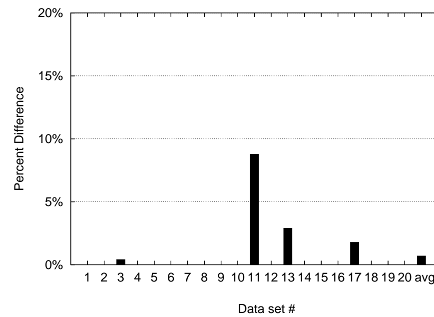

Fig. 10. *Average best opt relative to best opt(`tiff2rgba`).*

piler's baseline configuration. Fortunately, this occurs in only 6 of the 26 MiBench benchmarks we evaluated.

It is worth noting that programs that are related (i.e., encoders/decoders of the same audio/graphics formats or encryption/decryption algorithms for the same protocol) do not necessarily have similar sensitivity to data sets. For example, the encoding version of `rijndael`, has little variability with respect to the best performing optimization configuration across data sets. In contrast, its decoding version, has large variability in the performance of the best configuration across data sets.

3.4 Can iterative optimization perform well when training is done over multiple data sets ?

We now see if iterative optimization can perform well if we train over multiple data sets. It will also show whether it is possible to find an optimization configuration that is better on average over all the data sets of a program than the highest level of optimization available in the compiler.

For that purpose, we implement again a simple iterative optimization strategy which finds the "best average" optimization configuration across all data sets. For each data set, we compute the speedup achieved with each optimization, and determine which optimization performs best across all data sets on average. This strategy roughly emulates an iterative optimization process running for a long time, and finding the best optimization configuration compromise. We then compare this "best average" configuration against the actual best configuration for each data set.

Figures 9 and 10 depict the results of our analysis for two programs. Each graph shows the percent difference of the best optimization configuration found for each data set relative to the best average configuration across all data sets. Interestingly, the best average configuration across all data sets performs extremely well, typically within less than 5% of the best optimization configuration for each data set. This again has very positive implications for practitioners. We show in Section 3.3 that, for certain programs, it is important to not use one data set when performing iterative optimization. In that section, we showed that the best optimization configuration for any one data set can lead to potential degradation in performance when used with other data sets. Here we show that a possible answer is to collect various data sets for programs with high

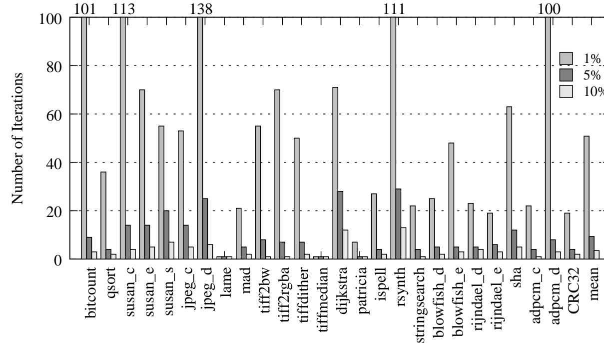

Fig. 11. Number of Iterations to get within 1%, 5%, and 10% for all benchmarks.

sensitivity to data sets and choose an optimization configuration that performs well on average over these data sets.

3.5 How fast can iterative optimization converge across multiple data sets ?

In this section we evaluate the difficulty of quickly finding good optimization configurations using iterative optimization across multiple data sets, and more precisely of getting close to the "best average" optimization mentioned in Section 3.4. For that purpose, we iteratively pick random optimization configurations and apply them to all data sets until the performance of the configuration is within a certain target percentage threshold of the best average configuration found previously.

Figure 11 shows the average number of iterations needed to find optimizations that are within 1%, 5%, and 10% of the best optimizations we found from a large exploration of the space. The figure shows that with a relatively small number of evaluations (usually around 5 evaluations) we obtain an optimization configuration that is within 10% of the best configuration found. Interestingly, if we want to be within 5% of the best optimization configuration we found, for most programs, it requires on average less than 10 evaluations. For only one program, dijkstra, is it a bit harder to obtain a configuration within 5% of the best configuration we found. This program requires on average evaluating around 30 configurations. Still, overall, relatively very few evaluations of optimization configurations are required to get much of the benefit of expensive iterative optimization. As one would expect on average it takes a much longer number of iterations to get within 1%.

3.6 In practice, is it possible to compare the effect of two optimizations on two data sets ?

Up to now, we have assumed that we can compare the impact of two optimizations and/or two data sets. However, in practice, we cannot compute the speedup due to an optimization because we would need two executions of the same data set: one to compute the execution time of the baseline optimization, and another run to compute the execution time of the optimization itself. As stated earlier, a user is unlikely to run the same program on the same data set twice. Instead we would like to consider IPC as a means of comparing the performance of two different optimization configurations on

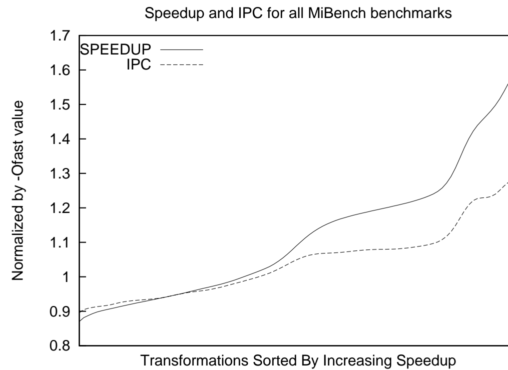

Fig. 12. *Speedup and IPC for all MiBench benchmarks (normalized by -Ofast value).*

different runs and data sets. IPC is an absolute performance metric and thus potentially does not require a base line run to measure improvement in contrast to speedup. However, some optimizations, such as loop unrolling, can effect dynamic instruction count which, in turn, affects the IPC.

Our experiments show that for most of the MiBench benchmarks (except two) the best found option corresponds to the best found IPC. Figure 12 shows the average speedup and IPC for all benchmarks and data sets for the MiBench benchmark relative to using PathScale's -Ofast configuration. This result is good news for iterative optimization as it means that we can use IPC to evaluate the effect of two optimization configurations regardless of the data sets used.

However, we note that the observations in this section may not hold for some architectures and with other compiler optimizations not evaluated in this study. In future work, we plan to enhance the technique of comparing the effect of optimizations using additional metrics to IPC and without the knowledge of the baseline execution time and the data set.

4 Emulating Continuous Optimization

If iterative compilation becomes mainstream, compilers will continuously optimize programs across executions with different data sets. After each execution, a program's performance results will be analyzed and a new optimization configuration to evaluate will be selected and the code will be optimized for the next execution. Some recent publications [19, 16] describe run-time continuous optimization frameworks but do not evaluate the influence of various datasets on optimizations and performance.

We leverage the observations of previous sections and attempt to emulate continuous optimization under as real conditions as possible. In other words, we assume it is not possible to know the speedup brought by a given optimization over baseline for a data set, only IPC is available to understand the effect of an optimization, data sets are executed in any order, and for each data set a different optimization is applied.

In addition, we use a simple strategy for selecting optimization configurations continuously. Each time an optimization configuration is tried on a data set, the IPC is recorded, the average observed IPC for that configuration is updated, and the configura-

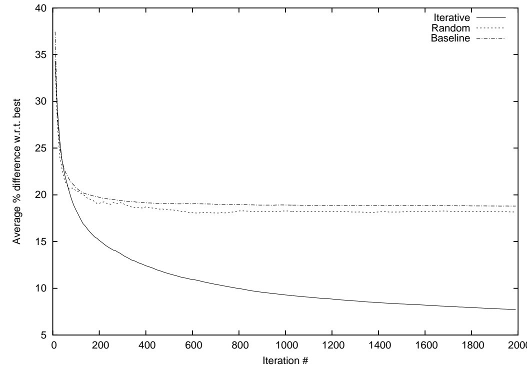

Fig. 13. *Results of continuous optimization experiment.*

tions ranking is then updated accordingly. Upon each iteration, configurations are randomly picked among the top N_{top} configurations (or we progressively bias the choice towards the current best known if not all optimizations have yet been tried; we use $N_{top} = 10$). In order to periodically revisit this ranking and adapt to new data sets, only 95% of the configurations are in fact picked among the N_{top} top, the remaining 5% are picked among all 200 configurations; the iterations where this occurs are themselves randomly picked according to the 95/5 uniform distribution.

Because continuous optimization is bound to occur over many iterations, our number of data sets is still too limited, so they have to be executed multiple times, but Section 3.2 suggests that there are not as many behavior as the number of data sets. Therefore, iterating over a collection of data sets is a much more reasonable approximation than iterating over a single data set.

Figure 13 shows the average difference with respect to the best possible configuration over the different iterations; this difference is itself averaged over all benchmarks, so the point at iteration `i` corresponds to the average difference over all benchmarks at that iteration. In addition to `iterative` compilation, the performance of a `random` strategy (configurations randomly picked according to a uniform distribution) and the `baseline` optimization are provided. The clear convergence of the iterative compilation process towards the best possible configuration shows that it is possible to learn the best optimizations, across data sets, and without any artificial means for evaluating the effect of optimizations.

In this section, no attempt was made to speed up the convergence, hence, our future work will take advantage of the existing literature on optimization search techniques [10, 11].

5 Related Work

Berube *et al.* construct a system called *Aestimo* which allows experimentation of feedback-directed optimization using different inputs [5]. The authors collect a large set of additional inputs for 7 SPEC benchmarks and study the variability of different inputs on inlining. They use the Open Research Compiler (ORC) and use profile-directed input

to control inlining. However, even though several data sets were collected for 7 benchmarks results and analysis in the paper are only presented for bzip2. The results show that the choice of inputs from which to generate profile information and control optimizations can have significant performance variability when used with different inputs. In contrast, we find that input variability largely depends on the benchmarks and/or the optimizations used, but for the most part most programs exhibit highly stable performance across data sets.

Bartolini *et al.* develop a software-based approach, called CAT, that repositions code chunks to improve cache performance [4]. The approach is profile-driven, that is, given an instrumented run of the program, a profile is collected which contains precise detection of possible interference between code chunks. The profile is then used to place code into memory in a way that reduces i-cache miss rates. Since the layout of the code is a function of the program input used during profiling, the authors evaluate different inputs and execution conditions to see the effect on input sensitivity. The authors evaluate different inputs on four multimedia application (jpeg encoder/decoder and mpeg encoder/decoder) and find that even though there is variability between inputs, their code placement technique is robust to this variability. These two papers are steps in the right direction to evaluating input sensitivity among programs, but they only present results for one optimization applied to a limited set of benchmarks and therefore general conclusions about data sets, applications, or optimizations cannot be drawn. In contrast, we look at the effects of many different optimizations applied to an entire benchmark suite using a larger number of data sets.

Haneda *et al.* attempt to investigate the sensitivity of data inputs to iterative optimization [15]. They perform iterative optimization using the `train` inputs on seven SPEC benchmarks. They perform iterative optimization using GCC 3.3.1 and control 42 of its options to find the best configuration for these benchmarks. They use a technique called Orthogonal Arrays to perform iterative optimization since it allows them to quickly find optimization flags that work well for a particular program. They obtain the best optimization flags using the train inputs for each benchmark and then apply this configuration to the benchmarks using the ref inputs. The authors find that the best optimization configuration found using train works well when applied to ref inputs.

Kulkarni *et al.* describe a partially user-assisted approach to select optimization sequences for embedded applications [18]. This approach combines user-guided and performance information with a genetic algorithm to select local and global optimization sequences. As is usually the case with iterative optimization papers, the authors do not consider input sensitivity and only show results for fixed benchmark/data set pairs. Other authors [13, 6] have explored ways to search program- or domain-specific command line parameters to enable and disable specific options of various optimizing compilers. Again, these authors keep the benchmark/data set pairs fixed.

Machine learning predictive modelling has been recently used for non-search based optimization. Here the compiler attempts to learn a good optimization heuristic offline which is then used instead of the compiler writer's hand-tuned method. While this work is successful in speeding up the generation of compiler heuristics, the performance gains have been generally modest. Stephenson *et al.* used genetic programming to tune heuristic priority functions for three compiler optimizations: hyperblock selection, register

allocation, and data prefetching within the Trimaran's IMPACT compiler [22]. The authors construct their heuristic on a training data set and report results on both a training data set and a test data set for the different optimizations. Using a data set different from the one used for training causes some degradation and in some cases dramatic reduction in performance. This may be due to the genetic algorithm "over-fitting" or specializing to the data set being trained on.

Eeckhout *et al.* attempt to find a minimal set of representative programs and inputs for architecture research [9]. They cluster program-input combinations using principal-component analysis (PCA) of dynamic program charateristics, such as cache misses and branch mispredictions. They find that while different inputs to the same program were often clustered together, in several cases different inputs to the same program result in data points in separate clusters.

MinneSPEC [21] is a set of reduced inputs for various SPEC CPU2000 benchmarks. They are derived from the reference inputs using a variety of techniques, such as modifying inputs (e.g., reducing number of iterations). They are often used to reduce simulation time for architecture research.

6 Conclusions and Future Work

In this article, we present a data set suite, called MiDataSets, for the MiBench benchmarks, which can be used for a more realistic evaluation of iterative optimization; this suite will be made publicly available. The scope of this suite extends beyond iterative optimization as many architecture research works are based on feedback mechanisms which are sensitive to data sets.

We use this data set suite to understand how iterative optimization behaves in a more realistic setting where the data set varies across executions. We find that, though programs can exhibit significant variability when transformed with several optimizations, it is often possible to find, in a few iterations, a compromise optimization which is within 5% of the best possible optimization, across all data sets. This observation, supported by a reasonable set of experiments (codes, optimizations, and data sets), has significant implications for the practical application and more widespread use of iterative optimization. Especially in embedded systems, but in general-purpose systems as well, the possibility that iterative optimization comes with great performance variability has slowed down its adoption up to now.

In future work, we plan on implementing a practical continuous optimization framework and enhance techniques of comparing the effect of optimizations without the knowledge about the data set or base line execution. This will help both validating the results of this article on a larger scale and in making iterative optimization more practical. We also plan to investigate the many different possible continuous optimization strategies on a fine-grain level, including for codes with high variability. We plan to enhance our MiDataSets based on program behaviour and code coverage. We also plan to evalute the influence of datasets on power and memory consumption which is critical for embedded systems.

References

1. PAPI: A Portable Interface to Hardware Performance Counters. `http://icl.cs.utk.edu/papi`, 2005.
2. PathScale EKOPath Compilers. `http://www.pathscale.com`, 2005.
3. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO-4: The Fourth Annual International Symposium on Code Generation and Optimization*, 2006.
4. S. Bartolini and C. A. Prete. Optimizing instruction cache performance of embedded systems. *ACM Trans. Embedded Comput. Syst.*, 4(4):934–965, 2005.
5. P. Berube and J. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006.
6. K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, 2001.
7. K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 69–77, 2005.
8. K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
9. L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-level Parallelism*, 2003.
10. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
11. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.
12. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
13. E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, 2001.
14. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.
15. M. Haneda, P. Knijnenburg, and H. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, 2006.
16. T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.
17. T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *The International Conference on Parallel Architectures and Compilation Techniques*, pages 237–248, 2000.
18. P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 171–182, 2004.
19. C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.
20. A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.
21. A. J. K. Osowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(2):10–13, June 2002.
22. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.
23. S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.
24. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.