Iterative Optimization for the Data Center

Yang Chen
SKL Computer Architecture *, China;
ICT and Graduate School, CAS, China
chenyang@ict.ac.cn

Shuangde Fang
SKL Computer Architecture*, China;
ICT and Graduate School, CAS, China
fangshuangde@ict.ac.cn

Lieven Eeckhout
Ghent University
Belgium
lieven.eeckhout@elis.UGent.be

Olivier Temam
INRIA, Saclay
France
olivier.temam@inria.fr

Chengyong Wu
SKL Computer Architecture*, China;
ICT, CAS, China
cwu@ict.ac.cn

Abstract

Iterative optimization is a simple but powerful approach that searches for the best possible combination of compiler optimizations for a given workload. However, each program, if not each data set, potentially favors a different combination. As a result, iterative optimization is plagued by several practical issues that prevent it from being widely used in practice: a large number of runs are required for finding the best combination; the process can be data set dependent; and the exploration process incurs significant overhead that needs to be compensated for by performance benefits. Therefore, while iterative optimization has been shown to have significant performance potential, it is seldomly used in production compilers.

In this paper, we propose Iterative Optimization for the Data Center (IODC): we show that servers and data centers offer a context in which all of the above hurdles can be overcome. The basic idea is to spawn different combinations across workers and recollect performance statistics at the master, which then evolves to the optimum combination of compiler optimizations. IODC carefully manages costs and benefits, and is transparent to the end user.

We evaluate IODC using both MapReduce and throughput compute-intensive server applications. In order to reflect the large number of users interacting with the system, we gather a very large collection of data sets (at least 1000 and up to several million unique data sets per program), for a total storage of 10.7TB, and 568 days of CPU time. We report an average performance improvement of 1.48×, and up to 2.08×, for the MapReduce applications, and 1.14×, and up to 1.39×, for the throughput compute-intensive server applications.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Design, Performance

Keywords iterative optimization, compiler, MapReduce, server, data center

* State Key Laboratory of Computer Architecture

ASPLOS'12, March 3–7, 2012, London, England, UK.

1. Introduction

Compilers embed a vast set of optimizations, implemented as separate passes of the global optimization strategy. Due to complex interactions among these optimizations as well as with the application software and the underlying architecture, it is exceedingly difficult to find the best combination and parameterization of compiler optimizations. The most aggressive default optimization level, e.g., `-O3` in the GNU C compiler, corresponds to a pre-set combination of compiler optimizations, with each optimization parameterized using simple analytical models or heuristics. However, it has been widely observed that there exist combinations of compiler optimizations that outperform the default optimization levels for many programs by a significant margin. This observation led to iterative optimization [1, 6, 8, 9, 13, 14, 20, 31], which is based on a simple but powerful concept: run a program multiple times, each time compiled with different combinations of compiler optimizations, in order to find the best combination. In spite of the great potential offered by iterative optimization and the significant amount of research done in this area over the past decade, it is still not widely used in (both commercial and public) production compilers, because it is plagued by at least three practical hurdles. (1) In order to find the best combination of compiler optimizations for a given program, many so-called recompilations (for different combinations of compiler optimizations) and training runs (running the recompiled binaries) are required. (2) The costly overhead of recompilation and training runs can easily wipe out the performance benefits of iterative optimization. (3) In most iterative optimization studies, the training runs are performed using the same data set(s), even though, in practice, there is no point for a user to run the same data set multiple times.

In this paper, we demonstrate that servers and data centers offer a context in which all of these hurdles can be overcome, paving the way for practical use of iterative optimization in the data center. We propose Iterative Optimization for the Data Center (IODC), a framework which, to the best of our knowledge, is the first to implement the idea of iterative optimization for the server and warehouse-scale computing domain. The key idea behind IODC is to carefully manage a cost versus benefit trade-off: IODC compensates for the cost of recompilations and training runs through the performance benefist of better performing combinations of compiler optimizations. The strategy operates automatically in the background, and is transparent to the end user, i.e., the user is not aware that training runs nor recompilations occur in the background.

We demonstrate the general applicability of IODC for both MapReduce-style as well as contemporary throughput compute-

intensive server applications. We find the MapReduce framework [11] to be a natural fit for IODC. The basic idea is to distribute different combinations of compiler optimizations across *workers*, which perform the recompilations and training runs, and which send performance statistics back to the *master*, which then in turn determines which combination yields the best performance. This process is done repeatedly until the system evolves to the best combination of compiler optimizations. Our experimental results using five MapReduce applications demonstrate an average speedup of 1.48×, and up to 2.08×.

As a second step in the paper, we extend IODC towards non-MapReduce, contemporary throughput compute-intensive server applications. The basic idea here is that the IODC run-time continuously applies the iterative optimization strategy as the program is repeatedly run on the server, possibly over long periods of time. Considering six benchmarks representative of throughput compute-intensive server applications, our strategy achieves an average speedup of 1.14×, and up to 1.39×.

Next to proposing and implementing iterative optimization for the data center, we believe this paper is the first to create realistic conditions for the evaluation of iterative optimization by avoiding data set reuse. We emulate data center operation by considering unique data sets across successive recompilations and training runs, which reflects many independent users interacting with the data center, all leading to different input data sets. In particular, for the MapReduce applications, we consider a very large number of unique data sets, varying between 1.55 million and 310 million unique records; for the server applications, we consider between 28,000 and 177,000 unique data sets, and we use each data set only once through the exploration process. By doing so, we emulate distinct production runs in a real data center.

In this paper, we make the following contributions:

- We propose iterative optimization for the data center (IODC). We believe that this paper is the first, to the best of our knowledge, to implement iterative optimization for warehouse-scale computing.
- We propose a novel online iterative optimization strategy that balances overhead versus benefit, alike a savings account. According to our experimental results, IODC achieves average performance improvements of 1.48×, and up to 2.08×, for a set of MapReduce applications, and 1.14×, and up to 1.39×, for a set of contemporary throughput compute-intensive server applications.
- We emulate realistic conditions in the experimental setup by considering unique data sets, and we demonstrate that IODC can learn good combinations of compiler optimizations across unique data sets. We believe this is the first paper to do so.

The remainder of this paper is organized as follows. In Section 2, we illustrate the performance potential of iterative optimizations and the challenges for applying iterative optimization in practice. In Section 3, we present the IODC strategy. This is done in two steps: we first implement IODC within the MapReduce framework, and we then extend the strategy towards contemporary throughput compute-intensive server applications. After explaining our experimental setup in Section 4, we present the evaluation in Section 5. Finally, we discuss related work in Section 6 and we conclude in Section 7.

2. Motivation

Iterative optimization is a compiler technique that can be applied to any program. Hence, it has the potential of being widely applicable and having broad impact. In particular, given that servers and data centers run the same set of workloads over and over again for many different users, improving performance and/or efficiency by only a small margin may lead to significant performance, energy and cost gains. However, successfully applying iterative optimization in practice is non-trivial. In order to find a good combination of compiler optimizations, many combinations may need to be evaluated because of the complex optimization space. Moreover, for some programs, the best combination may vary across data sets, making it even harder to find a good 'average' combination. Finally, in spite of its high performance potential, the benefits of iterative optimization are not sufficient to tolerate the overhead of a massive number of recompilations and training runs, hence, a careful trade-off between recompiling and training runs versus enjoying performance speedups from iterative optimization is required. In this section, we illustrate and quantify the most important challenges in iterative optimization.

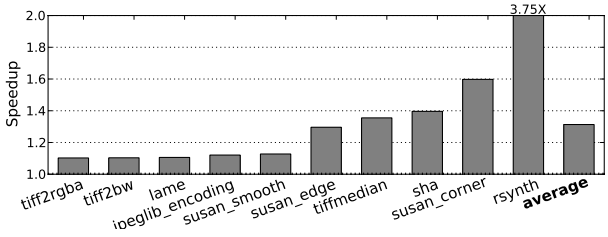

Figure 1. *The performance potential of iterative optimization: The average best performance speedup across 1000 data sets, relative to GCC* `-O3`.

2.1 Iterative optimization: Terminology

However, before doing so, we first introduce some terminology related to iterative optimization. As stated before, iterative optimization tries out a number of combinations of compiler optimizations to find out the best possible combination, or at least a combination that outperforms a compiler's best default optimization level, e.g., GCC's `-O3`. Trying out a combination involves compiling the program with the combination of optimizations, which we refer to as a *recompilation*, and running the compiled binary on real hardware with a given data set, which we refer to as a *training run*. We refer to the *evaluation* of a combination as the process of recompiling, doing the training run, and comparing performance against the compiler's best default optimization level. As will become clear later in the paper, evaluations are done in a way transparent to the end user. To make a distinction between training runs that drive iterative optimization on the one hand, and runs of a program that actually produce user output on the other hand, we refer to the latter as *production runs*.

2.2 Potential of iterative optimization

The performance potential of iterative optimization is substantial, as we will illustrate with the following experiment. We consider a set of 300 combinations, 10 benchmark applications, each with 1000 distinct data sets, and we apply all combinations to all data sets for all benchmark applications, 3 million runs in total. (We refer to Section 4 for a detailed description of the experimental setup.) Then, for each benchmark application and each data set, we select the combination that provides the best performance; subsequently, for each benchmark application, we compute the average best performance across all data sets. The results are reported in Figure 1; the benchmarks are ranked on the horizontal axis by increasing average best performance. The speedup ranges from 1.10× to 3.75×, with an average of 1.31×. These results illustrate the maximum performance that can be achieved using iterative optimization if we had an oracle indicating, for each program and each input data set, what would be the best possible combination of compiler optimizations. The difficulty now is to achieve similar performance improvements in practice.

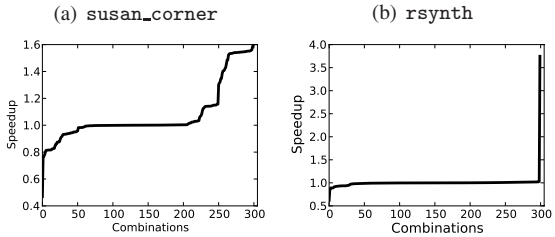

Figure 2. *Cumulative distribution of the average speedup for combinations of compiler optimizations ranked by increasing speedup for (a) `susan_corner` and (b) `rsynth`.*

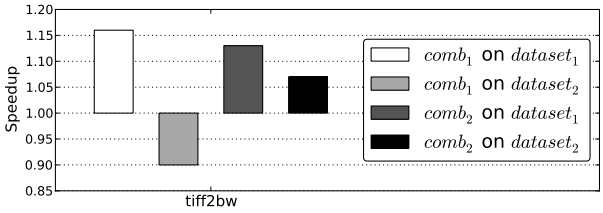

Figure 3. *Demonstrating data set sensitivity for iterative optimization: Applying combinations selected using a data set to another data set for `tiff2bw`.*

2.3 Iterative optimization requires many runs

For some programs, the number of good combinations can be rare, and thus difficult to find, requiring a possibly large number of combinations to evaluate in order to identify the best possible combination(s). Figure 2 illustrates that the number of 'good' combinations varies significantly across programs. For each combination, we average the speedup over 1000 data sets, and we then rank the combinations by increasing average speedup. For `susan_corner`, more than 15% of the combinations yield a speedup of at least 5%, and a few combinations even yield a speedup of 60%. For `rsynth` on the other hand, less than 0.4% of the combinations yield a substantially large speedup. In other words, only a few combinations yield the best possible performance, hence, finding them may require a large number of recompilations and training runs.

2.4 Data set sensitivity

Not only do different programs favor different combinations, also different data sets for the same program may favor different combinations. We illustrate this point in Figure 3 for `tiff2bw`, and two data sets $dataset_1$ and $dataset_2$. For each of these two data sets, we determine the best combination among a set of 300 combinations. $comb_1$ is the best combination for $dataset_1$ (see '$comb_1$ on $dataset_1$'); $comb_2$ is the best combination for $dataset_2$ (see '$comb_2$ on $dataset_2$'). We subsequently apply each combination to the other data set. In some cases, the same combination works (fairly) well for more than one data set, see '$comb_2$ on $dataset_1$' versus '$comb_2$ on $dataset_2$'. However, in other cases, a combination that performs best for one data set may even induce slowdowns for other data sets, see '$comb_1$ on $dataset_2$' versus '$comb_1$ on $dataset_1$'. These results suggest that it is challenging to find a combination that performs well on average across a broad set of data sets. Also, it is to be expected that the 'average' data set changes over time as users come and go and solve different problems. Hence, the best combination that performs well on the average data set must be carefully selected, and the choice must be constantly revisited.

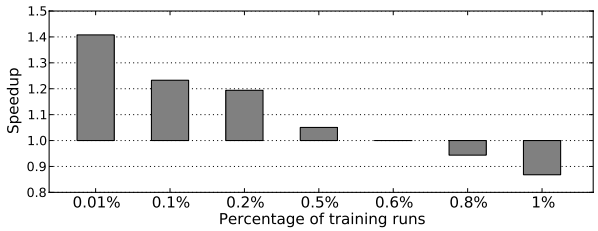

Figure 4. *Impact of training runs on the strategy for `ann`.*

2.5 Overhead/benefit trade-off

In order to explore and compare different combinations of compiler optimizations, programs must be recompiled followed by a training run. While the benefits of iterative optimization can be significant, as demonstrated in the previous sections, they are not sufficient enough to compensate for a careless use of training runs and recompilations. In fact, in order to get any overall performance benefit from iterative optimization, it is essential to find the best possible trade-off between exploring combinations and minimizing training runs and recompilations. In Figure 4, we apply our proposed IODC strategy and vary the total number of evaluations (recompilations plus training runs) as a percentage of the total number of production runs, from 0.01% to 1% for `ann`. Even 0.6% of training runs are enough to wipe out any benefit from iterative optimization, and 1% of training runs results into a slowdown over no iterative optimization by 13%.

2.6 Summary

In conclusion, iterative optimization has significant performance potential, however, achieving it in practice is non-trivial for a number of reasons. (1) A potentially large number of combinations of compiler optimizations may need to be evaluated. (2) Iterative optimization can be data set dependent. (3) A careful balance between benefits and costs is required for iterative optimization to yield overall performance benefits in practice. The data center provides a context in which each of these challenges can be addressed, as will become clear in the next section.

3. Iterative Optimization for the Data Center

We now introduce Iterative Optimization for the Data Center (IODC) which can be applied to both MapReduce-style workloads as well as throughput compute-intensive server workloads. We first introduce IODC for MapReduce workloads, followed by IODC for throughput compute-intensive server workloads.

3.1 MapReduce vs. iterative optimization

MapReduce is a paradigm and framework for processing huge data sets on a distributed system [11]. MapReduce splits up computation in a Map and a Reduce stage. In the Map stage, the *master* partitions the input into smaller sub-problems, and distributes those to the *workers*. Each worker node works on a sub-problem which in itself consists of a number of so-called *records*. In our setup, the entire problem consists of millions of records which are partitioned across the workers in sub-problems, each consisting of on the order of at least thousands of records. The *Map function* represents the work done by the worker in the Map stage; the worker nodes produce intermediate files. In the Reduce stage, the master assigns reduce tasks to the worker nodes, which then produce the final output from the intermediate files. The *Reduce function* specifies what a worker needs to do during the Reduce stage.

This general concept and organization seems to be a natural fit with how iterative optimization operates. Iterative optimization requires that a program be executed many times, albeit compiled differently, in order to understand which combination of compiler

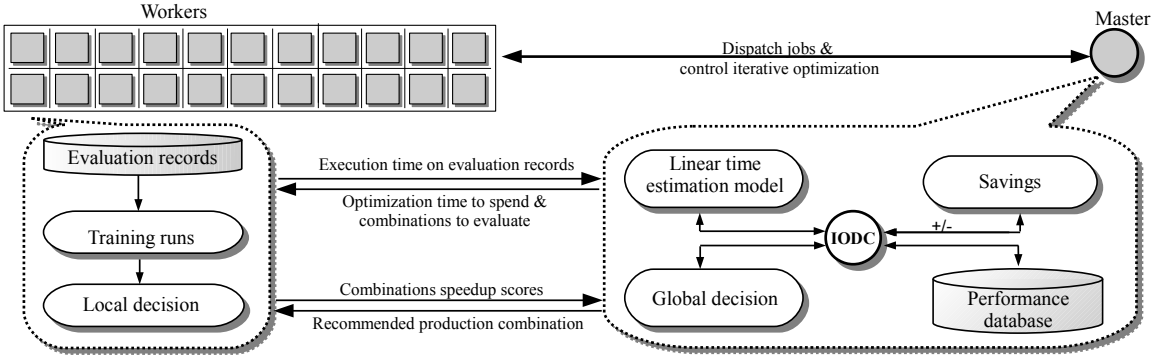

Figure 5. *IODC for MapReduce workloads.*

optimizations yields optimum performance. In MapReduce terminology, the master is running the iterative optimization strategy, while the workers evaluate different combinations of compiler optimizations (recompilations and training runs), and send performance statistics back to the master. Based on the aggregated performance statistics, the iterative optimization strategy running on the master then propagates recommendations to the workers about which compiler combinations to employ in the future.

3.2 IODC for MapReduce

Although the overall concept of IODC seems to be an intuitive fit for MapReduce workloads, designing an efficient strategy is challenging. The main difficulty is to make sure that the overhead of training runs, recompilations and communication overhead between the master and the workers do not outperform the benefits of iterative optimization. The key principle of our approach is to manage costs and benefits carefully. As mentioned before, evaluations of combinations of compiler optimizations are done on-the-fly, in a way that is transparent to the user. Further, iterative optimization continuously searches for better performing combinations of compiler optimizations. At any point in time, IODC employs the combination that yields the best possible performance among the combinations evaluated so far. We refer to the current best combination as the *production combination*, as it is the one currently being employed and producing output for the end user. At the same time, IODC evaluates other possible combinations in the background, hoping to identify a combination that yields even better performance. The strategy initiates with the default `-O3` optimization level as the current production combination, and evolves towards the best possible combination over time.

IODC's strategy uses a novel algorithm for managing costs and benefits from iterative optimization, much alike a 'savings account'. IODC starts off with a zero balance on the savings account, and for each record for which the selected combination of compiler optimizations performs better than the default compiler optimization, we consider that we achieve an execution time 'benefit', increasing our savings. A training run and recompilation 'spends' some of the earned benefits and decreases the balance on the savings account.

Figure 5 illustrates the general flow of IODC for MapReduce workloads. The master dispatches jobs to the workers, i.e., the master gets the workers to execute the Map and Reduce functions on a particular subset of the records. The master also controls iterative optimization, and keeps track of a performance database of previously evaluated combinations of compiler optimizations as well as the 'savings account' for managing iterative optimization overhead. We will describe IODC in more detail in the following paragraphs.

Sampling records for evaluating combinations. The master distributes records among the workers in a MapReduce workload. IODC will in addition direct the workers to randomly select a sample of its assigned records, and to use these as *evaluation records*. On all these evaluation records, the workers run the Map or Reduce functions, compiled with the default compiler optimization (i.e., `-O3`). In addition, the workers will also recompile the Map and Reduce functions and perform training runs using the evaluation records. The workers can then determine locally, among all the combinations evaluated, which one is the best. Then, they propagate this information back to the master for making a global decision on the best combination, see also Figure 5. The evaluation records are also used to detect program errors caused by compiler bugs during recompilation; we will discuss this further in Section 3.5.

Estimating the performance gain by a combination. The strategy spends only a fraction of the savings brought by good combinations, in order to achieve a net benefit at any time during the MapReduce run. This objective can only be met if the savings are properly evaluated. One way to accurately evaluate the performance benefit due to a combination is to execute each record twice, using the default compiler optimization and the other combination, which would introduce too high overhead. Instead, we estimate the average performance gain using the evaluation records only, not all records. For each of the evaluation records, the workers evaluate the speedup brought by the new combinations over the default compiler optimization, and send these results back to the master. The master uses this information and the knowledge on the total remaining number of records to estimate the new savings that could be achieved with the new anticipated production combination. This is done through linear extrapolation, referred to as the linear estimation model in Figure 5.

Based on the anticipated savings, the master decides how many combinations can be evaluated by the workers, and directs them to perform the corresponding number of evaluations. The principle is to 'invest' only a small fraction (P%) of the total savings in order to avoid recklessly spending the savings, we use P=20% in our setup. The evolution of savings is illustrated in Figure 6. Initially, with no savings, we invest only a small fraction of the total estimated workload execution time (i.e., 1% in our experiments) to do explorative evaluations (recompilations and training runs). Once better combinations are found, only P% of savings brought by these combinations are invested to initiate more evaluations for finding even better combinations. As a result, we keep most of the savings. Note that, in either case, as long as a program is repeatedly executed, explorative evaluations will continue. Better combinations will eventually be found even if they are rare in the optimization space as in the case of Figure 2(b). In the worst case,

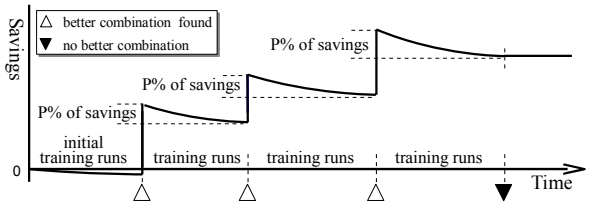

Figure 6. *Evolution of savings.*

i.e., if no better combinations are found, only a negligible loss is incurred.

Revisiting combinations vs. evaluating new combinations. As explained above and shown in Figure 5, the master will periodically allocate a certain amount of savings to workers for exploration purposes. The workers perform the explorative evaluations in parallel. Each worker evaluates one or a few combinations. The goal of the exploration is twofold: (i) to find new combinations that may perform better than the current best optimization, but also (ii) to revisit previous production combinations. The reason for both looking for new combinations and revisiting combinations, is to be able to continuously find the best possible optimization, even in the presence of time-varying inputs.

The master distributes to each worker a mix of new combinations and known combinations. As soon as the evaluations are performed, the workers send the speedup score of each evaluated combination to the master, see Figure 5 (top arrow from the workers to the master), which maintains a probability distribution characterizing the quality of each combination, based on the speedups recorded for that combination. The master selects the best combination known so far as the production combination, and propagates it to all the workers, see Figure 5 (bottom arrow from the master to the workers).

Within-run IODC vs. across-run IODC. IODC can be applied both within a single run and across runs. Within a MapReduce run, a large number of Map and Reduce operations are dispatched by the master to the workers, i.e., each worker operates on a set of assigned records. Once IODC finds a better combination than the current production combination, this better combination becomes the new current production combination, and it is immediately used to process the remaining records. This is referred to as within-run IODC. Well-performing production combinations can also be used across multiple runs of the same MapReduce workload (e.g., by different users), and IODC can continue to optimize the production combination across runs. Across-run IODC maintains optimization information across runs, which allows for continuously optimizing the production combination.

3.3 Extending IODC to any server application

While MapReduce is well suited for implementing iterative optimization because of the large number of records that need to be processed in parallel, the concept developed so far can be extended to any application running on a server. Contemporary throughput compute-intensive server applications (i.e., non-MapReduce batch-style workloads) do not typically run many instances of the same application at the same time, as is the case for MapReduce workloads. However, over a sufficiently long period of time, a similarly large number of runs are performed with different data sets for each run.

We therefore develop a modified iterative optimization strategy that does not rely on multiple simultaneous runs and a master/worker relationship, but which instead stores a *history* of combinations of compiler optimizations for each program. The strategy is implemented as a standalone script, running on one of the server nodes; the overhead of running the strategy script is factored in our evaluation, though it is actually negligible. As shown in Figure 7, the modifications to the strategy are the following.

Sample data sets. We maintain a pool of D randomly selected data sets, which we use the same way as the evaluation records for the MapReduce workloads: we use these evaluation data sets to evaluate the performance of alternative combinations of compiler optimizations. This pool is periodically refreshed as new data sets come in, in order to adapt to the possibly changing nature of data sets over time.

D has to be kept small, lest the overhead of the comparison is too high, i.e., $D = 3$ is the default value of the strategy in our setup. Because D is small, the comparison between the default and an alternative combination of compiler optimizations may be inaccurate so we check whether the mean execution times (over the D data sets) are statistically significant. For that purpose, we use the t-test with resampling [27]. If the comparison shows that the new combination would bring a performance improvement of at least 1% at a 95% confidence level, we replace the current production combination with the new combination.

Conservative and aggressive modes. Unlike in MapReduce, the strategy cannot estimate upfront the total savings brought by a better combination because the total number of runs of the program (over its entire lifetime on the server) is unknown. Therefore, exploration decisions have to be revisited after each run. The amount of exploration the strategy can afford to perform at any given time depends on the accumulated savings so far, and upon the rate at which better combinations are found. The latter criterion is the motivation for two modes: if better combinations are frequently found, future savings and the promise of even greater savings justify to quickly search for new combinations; otherwise, exploration must proceed more carefully.

The strategy toggles automatically between a 'conservative' and an 'aggressive' mode. The only distinction between these modes is the percentage P of savings that we are ready to spend. We use $P = 5\%$ in conservative mode, and $P = 50\%$ in aggressive mode. This percentage does not apply to all cumulated savings since the beginning of the task, but only to the cumulated savings since the last mode change. As a result, all savings accumulated till the end of a mode are definitely won.

Consider the example of Figure 8. Initially, the strategy starts in aggressive mode, speculatively investing future savings in order to quickly find better combinations. In this example, we assume that the exploration performed on the first data sets finds some better performing combination. The strategy will decide to stay in aggressive mode, postulating even better combinations can be found. After several more combinations being evaluated without showing an additional benefit, the strategy toggles back to conservative mode. No exploration will take place until the overhead of the last evaluations correspond to no more than 5% of the cumulated savings so far. When that happens, a few more combinations are explored. If that exploration is successful again, i.e., an even better combination is found, the strategy toggles back to aggressive mode, and so on.

3.4 Implementing IODC

We now discuss a number of IODC implementation issues which we find useful mentioning.

MapReduce. Implementing IODC in MapReduce requires that the worker nodes know where to find the Map and Reduce functions for recompilation. In our implementation, a new API was added so that the user can specify the location of the source code of the Map and Reduce functions for automatic recompilation. An alternative approach would be to use Unix environment variables for passing location information.

Capturing inputs and outputs. Iterative optimization uses recompilation and training runs to evaluate different combinations of compiler optimizations. In order to be transparent to the users, we need to automatically retain the inputs of evaluation runs and

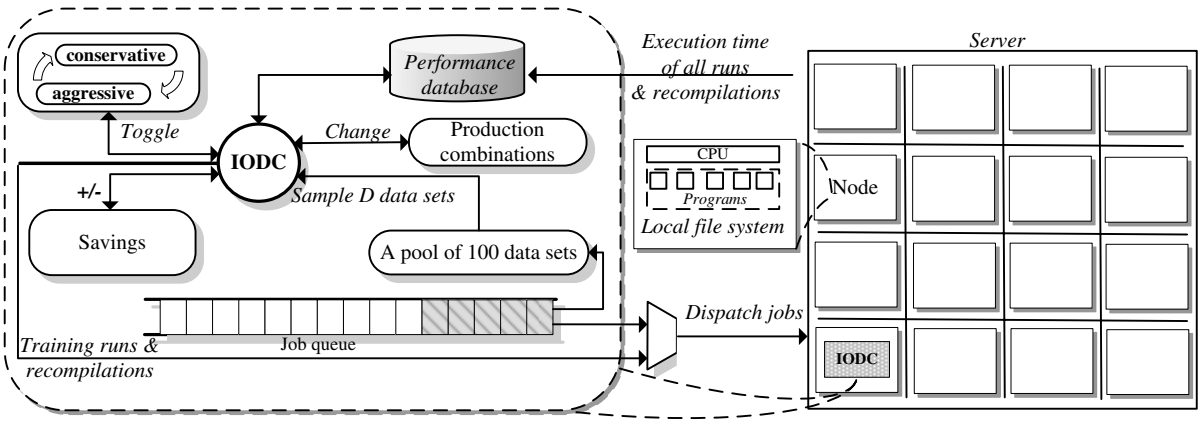

Figure 7. *IODC for throughput compute-intensive server applications.*

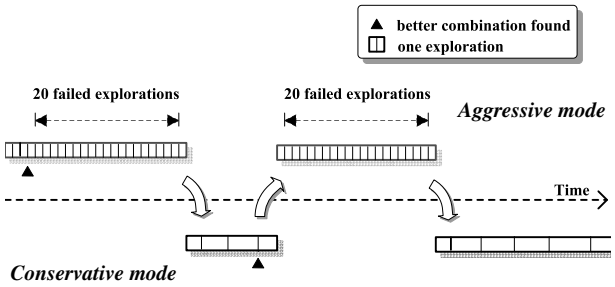

Figure 8. *Toggling between conservative and aggressive modes.*

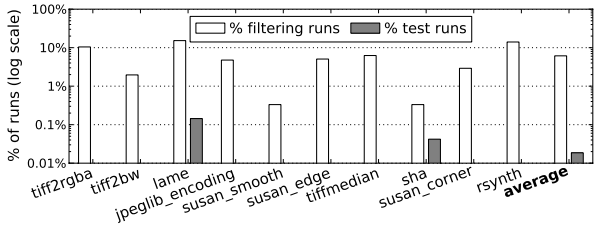

Figure 9. *Percentage of runs where a compiler bug was detected (1000 runs for each benchmark application).*

capture their outputs. In the MapReduce framework, the runtime system is in charge of capturing, transferring and storing the input and output data of user-defined Map and Reduce functions, while the user only needs to specify the locations of input and output through a set of framework-specific API calls. In the runtime, we intercept messages between the master and the workers in order to know the input location and sample them for evaluation runs. To reduce disk seek time, we sample the input at the granularity of a group of consecutive records, not individual records. We also modified the runtime to not transfer nor store the outputs of Map and Reduce functions during evaluation.

Propagating recompiled code. Once the current best combination gets promoted as the next production combination, we need to propagate this information from the master to the workers. In the MapReduce implementation, only the combination itself is sent to the other workers, which perform recompilations locally. In the server implementation, IODC sends the recompiled code across the nodes. The time required for either propagation method is factored in our performance measurements.

IODC for server applications. For retaining the D data sets in IODC for server applications, we leverage the replay strategy which is commonly used in large-scale servers for fault-tolerant computing [2, 23]. Because the failure rate in servers is high, software infrastructures on servers embed a replay capability to rerun a data set if the previous run did not complete. That capability requires to temporarily retain the data set, which we leverage in IODC.

Heterogeneous servers. So far, we have assumed that all the machines in a cluster are the same, so that results gathered on one node can be factored in for other nodes. It is straightforward to extend the strategy to factor in heterogeneous nodes: the information on which combinations perform best must only be partitioned into classes, each class corresponding to a node type or configuration.

3.5 Compiler bugs

The interplay among different compiler optimizations is particularly complex, up to the point where it is difficult for compiler designers to ensure that any combination of compiler optimizations will always result in bug-free programs. Therefore, we need to add a safeguard mechanism to check the correctness of code recompiled with complex combinations of compiler optimizations.

For that purpose, we leverage our evaluation records again and the aforementioned facilities for capturing output. The output of the evaluation records, compiled with the default compiler optimization, are stored. Whenever a new combination is evaluated on one of these evaluation records, the outputs are compared. If there is any difference (or if the run crashes for that record), the combination is flagged as potentially buggy and not authorized for production runs.

In order to illustrate how frequent the problem is, and to assess the efficiency of our safeguard mechanism, we proceed as follows. We generate random combinations of compiler optimizations for the GNU C compiler, and we emulate the safeguard mechanism by running each combination on 5 randomly selected data sets for

each of the benchmark applications; we repeat this process until we identify 300 fault-free combinations; the experimental framework is presented in more details in Section 4. The percentage of faulty runs is shown in Figure 9, see the '% filtering runs' bars. Any combination found to be faulty is no longer considered, it is deemed filtered by the safeguard. Then, in order to assess the efficiency of the safeguard, we run the remaining, filtered, 300 combinations on the 1000 data sets of our 10 benchmark applications, 3 million runs in total. The fraction of invalid runs is reported in Figure 9, see '% test runs'. While some compiler bugs were not captured by the filtering process, their occurrence is rare, 0.02% on average. We repeated the same experiment with a commercial compiler, Intel ICC, for which 3.3% of filtering runs were faulty, and for which we found no remaining compiler bugs after filtering.

4. Methodology

Before presenting the evaluation of IODC, we first describe our experimental setup.

4.1 Servers

We evaluate IODC on three different kinds of servers. The first one is a 7-node DELL cluster, with 3GHz Intel Xeon dual-core processors (E3110 family), 2GB RAM, 2x3MB L2 cache, running CentOS release 5.3 (Final). We refer to this cluster as the *Intel cluster* in the remainder of the paper. The second one contains 8 dual-core AMD Opteron processors (8218), 16GB RAM, 2x1MB L2 cache, running Red Hat Enterprise Linux 4.4. We refer to this cluster as the *AMD cluster*. The third one is a 32-node cluster with Loongson 2F processors [22], running Red Flag Linux 6.0; Loongson 2F is an 800MHz MIPS-compatible general-purpose CPU, with a 512KB L2 cache and 512MB RAM. We refer to this cluster as the *Loongson cluster*. In order to perform the very large number of runs described below in a reasonable amount of time, we partitioned our benchmarks across these three clusters as indicated in Table 1.

4.2 Benchmarks and data sets

We consider three groups of workloads for evaluating IODC, see also Table 1.

MapReduce applications. The first group of workloads is a set of five MapReduce applications: `minhash`, `ann`, `kmeans`, `knn`, and `pfsp`. `minhash` is a probabilistic clustering algorithm implemented as a MapReduce component in the Google News service [10]. `ann` is a back-propagation algorithm for training the weights of a three-layer artificial neural network [21]; `kmeans` and `knn` are MapReduce implementations of k-means clustering [28] and the k-nearest neighbor algorithm [17], respectively; `pfsp` relies upon a genetic algorithm and a NEH-based heuristic method [24] to solve the permutation flowshop scheduling problem, which is an important task in industrial engineering, aiming at finding a job sequence with minimal makespan [35].

The MapReduce implementation we use is Sector/Sphere [18], an open-source high-performance distributed file system and parallel data processing engine. We generated 1,550,000 to 310,000,000 inputs (records) for these tasks, either based on a uniform distribution, or a distribution extracted from real-world data, see also Table 1. We randomly grouped these records into 5 different data sets of the same size, for each of the applications. We will use these 5 data sets to evaluate within-run IODC versus across-run IODC.

Compute-intensive server applications. The second group of workloads contains six programs, five of which are extracted from the PARSEC benchmark suite [5]: `x264`, `ferret`, `freqmine`, `blackscholes`, and `canneal`; the sixth server application is `bzip2e`, the well-known file compression utility. These programs are similar to emerging compute-intensive server applications used in several popular web services. `x264` is an H264 video compression program, which is the kind of tasks run on YouTube servers, among others. `ferret` performs content-based similarity on images; Google Picasa and Flickr both rely on content-based similarity for identifying faces within images. `freqmine` is a typical data mining program aiming at finding frequent item sets; Amazon uses similar algorithms to recommend related books. `blackscholes` is an option-pricing prediction model, popular in financial institutions; it is used for instance by Morgan Stanley and Goldman Sachs. `canneal` is a chip routing application, similar to the place and route algorithms used by EDA companies (e.g., Xilinx). `bzip2e` is a compression algorithm, broadly used in file sharing web sites, such as the GNU FTP. For these six programs, we collect between 28,000 to 177,119 data sets, see Table 1, corresponding to a total storage of 9.54TB. In all experiments, we use each data set only *once*, thereby emulating distinct production runs.

Benchmark applications. The third group of benchmarks are a collection of ten relatively small applications, which are similar to tasks running in the backbone of popular web services. These benchmarks are taken from well-known open-source projects. We use these benchmark applications for fine-tuning IODC in a reasonable amount of time prior to running the more complex and long-running MapReduce and server applications.

We consider the following *benchmark applications*. `rsynth` performs text-to-speech conversion, as provided by the service `readthewords.com`; `susan_corner`, `susan_edge`, and `susan_smooth` (all from susan) perform standard and compute-intensive image processing steps of image recognition applications, as in `myheritage.com`; `jpeglib_encoding` (from jpeglib), `tiff2median`, `tiff2bw`, `tiff2rgba` perform image conversions, as provided by `converthub.com` or `go2convert .com`; `lame` is an mp3 encoder, as provided by `media.io`; `sha` is the SHA hashing algorithm, broadly used in servers [29].

Combinations of compiler optimizations. For all experiments, we use the GNU GCC v4.4 compiler, and the optimization level `-O3` as the baseline. We selected 127 compiler options of GCC that control inlining, unrolling, vectorization, scheduling, register allocation, constant propagation, among other optimizations known to have a potential impact on performance. We then create 300 randomly chosen combinations of these compiler optimizations. We explore these combinations in random order in the experiments. Note that other exploration methods [1, 6, 25], which may require extensive training and/or program feature extraction, can also be integrated into IODC.

4.3 Measurements

We measure end-to-end wall clock time, and we report speedups over the GCC `-O3` optimization level. The performance numbers reported in the paper include all sources of overhead, including the cost of recompilations, training runs, communication overhead between the master and the workers, etc. The total CPU time was 110 days for the MapReduce applications, 440 days for the server applications, and 18 days for the benchmark applications.

5. Performance Evaluation

We now evaluate IODC. This is done in a number of steps. We first evaluate IODC for MapReduce and server workloads. We subsequently evaluate IODC's impact on energy consumption, and we explore various optimizations of the IODC framework.

5.1 MapReduce

Figure 10 reports the speedup achieved by within-run IODC. Speedups range up to 1.91×, with an average speedup of 1.29×. There is a slight performance degradation for `kmeans` due to the initial 'investment' into the exploration of combinations which failed to find a better combination than GCC's default `-O3` optimization level, and which results in a 1.24% slowdown. Note though that we

Benchmark suite	Program (domain)	Data set number (total file size)	Data set description	Cluster
MapReduce applications	minhash (probabilistic clustering)	10830935 (6GB)	Generated data sets obeying distribution extracted from MovieLens[34]; each record represents a movie history seen by one user.	Loongson [22]
	ann (model training)	310000000 (222GB)	Generated data sets obeying uniform distribution; each record consists of input and output for the artificial neural network.	Loongson
	kmeans (distance-based clustering)	4294720 (320GB)	Generated data sets obeying uniform distribution; each record represents a multi-dimension point.	Loongson
	knn (query-based learning)	387500000 (580GB)	Generated data sets obeying uniform distribution; each record consists of an instance and its label.	Loongson
	pfsp (scheduling problem)	1550000 (591MB)	Generated data sets obeying uniform distribution; each record represents a job scheduling scheme.	Loongson
Server applications	bzip2e (compression)	28000 (438GB)	Uncompressed tar balls of source and binary packages of various Linux distributions.	Loongson
	ferret (image similarity search)	58430 (187GB)	Groups of query images constructed using 1,166,657 JPEG files downloaded from 130 different web sites.	Intel
	freqmine (data mining)	61655 (4.05TB)	Lists of transactions randomly generated using IBM Quest Market-Basket Synthetic Data Generator [3].	AMD
	x264 (video encoding)	67571 (2.28TB)	Yuv4mepg videos converted from MKV and RMVB files downloaded from the Internet.	Loongson
	canneal (electronic design automation)	177119 (2.15TB)	Netlists randomly generated with an enhanced version of the input generator provided by PARSEC.	Intel
	black- scholes (financial analysis)	43354 (434GB)	Lists of options randomly generated based on the value ranges deduced from the 1000 inputs provided by PARSEC.	AMD
Benchmark applications	10 benchmark applications	1000 each (10GB)	1000 data sets per benchmark collected from various web sites [7].	Intel

Table 1. *Benchmarks, data sets and servers considered in this study.*

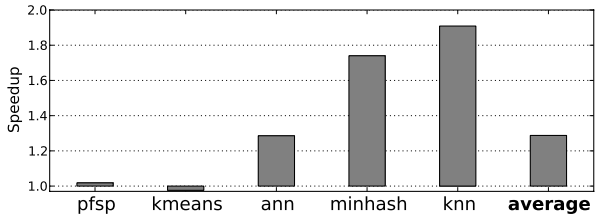

Figure 10. *Speedup for the MapReduce applications for within-run IODC.*

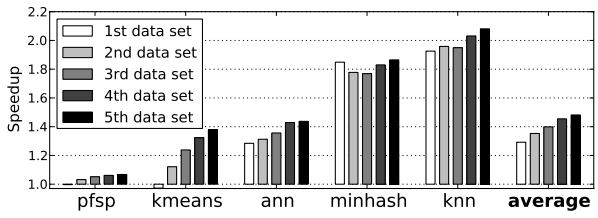

Figure 11. *Speedup for the MapReduce applications using both within-run IODC (1ste data set) and across-run IODC (2nd through 5th data set).*

perform within-run IODC only in this experiment, which explains the slowdown for `kmeans`.

To evaluate across-run IODC, we now consider five data sets to emulate five consecutive runs of the same MapReduce workload with different inputs, see Figure 11. On average, the speedup achieved across runs for the combined strategy is 1.48× after 5 data sets, versus 1.29× after 1 data set. The maximum speedup is 2.08×, achieved for `knn` after 5 data sets. Note that we now achieve a speedup of 1.38× after 5 data sets of across-run IODC for the `kmeans` which showed a 1.24% slowdown with within-run IODC.

5.2 Server applications

In Figure 12, we report the cumulated speedup for the server applications using unique data sets. The curves report how performance speedup improves over time. At any given point in time, the cumulated speedup is computed as the ratio of the wall clock time for all runs compiled with `-O3` versus all runs compiled with production combinations. The 'with overhead' curve accounts for all the overhead costs, whereas the 'no overhead' curve discounts any source of overhead; the delta between both curves quantifies the overhead.

We observe that, initially, when the strategy searches for good combinations, the cost of recompilations and training runs results in a slowdown. However, the corresponding slowdown is on the order of 1% on average. The initialization phase does not take that long; after a couple thousand runs (2670 runs on average), IODC yields a net performance benefit; the reason is that combinations that outperform the default `-O3` production combination can usually be found quickly. As shown in Figure 12, the proposed strategy achieves a cumulated speedup of 1.39× for `bzip2e`, 1.12× for `blackscholes` and `freqmine`, 1.10× for `x264`, 1.08× for `canneal`, 1.06× for `ferret`, with an average of 1.14×. We also find the overhead of IODC to be small, a few percent only.

Also note the thin marks at the bottom of each of the graphs in Figure 12. These marks indicate when the strategy searches for a new combination. Note that there is sometimes an inflection point after which a much better combination is found; the cumulated speedup stops increasing at a fast pace for a little while, and then after the inflection point, it starts increasing faster, see `ferret` for the most notable example. Right before the inflection point reflects the aggressive mode. Once a better combination is found, the strategy transitions back to conservative model, after which performance benefit increases again.

In the different graphs of Figure 12, performance plateaus after some time because we use a uniform random mix of data sets, so the strategy will eventually end up finding a program-optimal combination, i.e., a combination that performs well across many data sets. With a less uniform mix, the choice of the best combination has to be constantly revisited, which is described in Section 5.6.

Overall, these experiments show that, even under conditions with unique data sets and factoring in all the overhead costs, IODC improves the performance in a consistent way, i.e., without incurring performance degradation at any time except in the initial phase.

5.3 Energy reduction

We now quantify the energy implications of IODC. Given the importance of energy consumption on total cost of ownership (TCO), reduction in energy consumption immediately translates into proportional cost reductions. We use a simple linear model to estimate energy consumption based on CPU utilization, as proposed

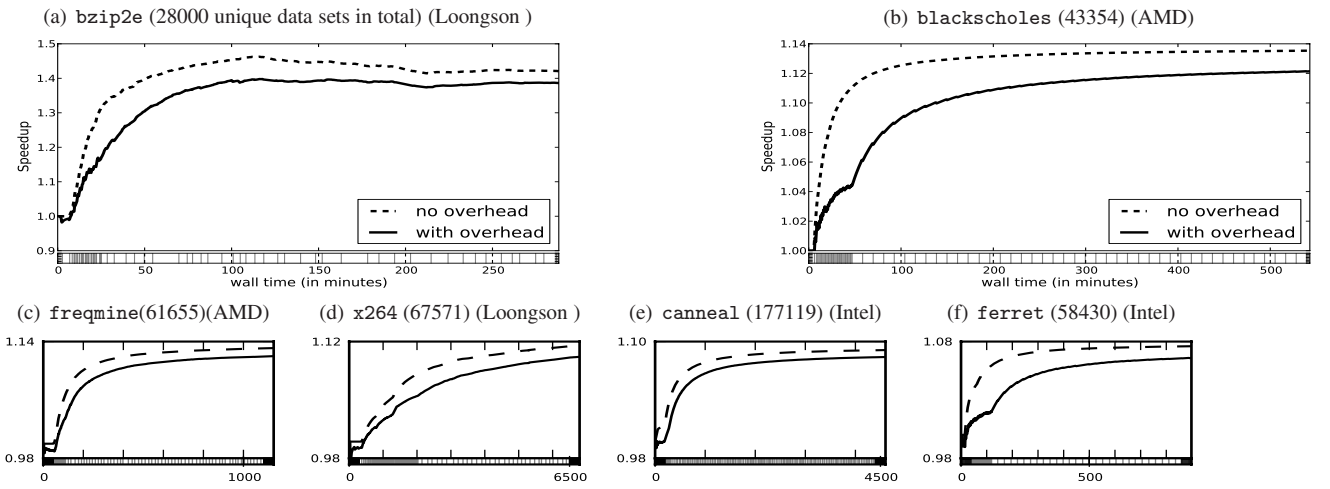

Figure 12. *Cumulated speedup profiles for* `bzip2e`*,* `blackscholes`*,* `freqmine`*,* `x264`*,* `canneal`*, and* `ferret`*.*

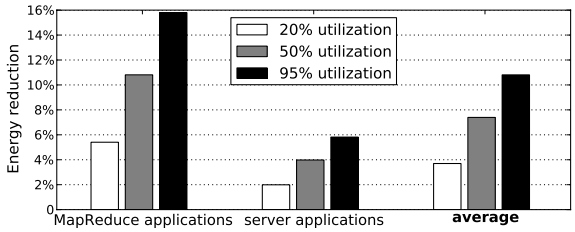

Figure 13. *Estimating energy saving through IODC at different system utilization levels.*

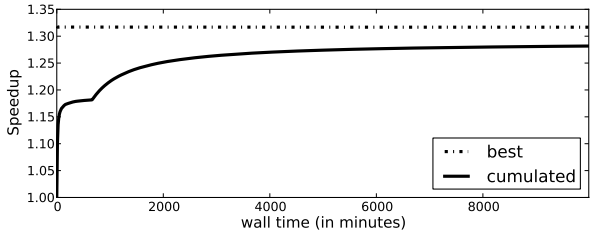

Figure 14. *Cumulated speedup profile averaged across the benchmark applications.*

by Fan [12]:

$$power = P_{idle} + (P_{busy} - P_{idle}) * u \quad (1)$$

where P_{busy} is the power of the server running the most power-intensive tasks; P_{idle} is the power consumption in idle state, and it is equal to half that of P_{busy} according to the results by Fan et al. [12]; u is CPU utilization. We consider three different utilizations: 20%, 50% and 95%. We report the results in Figure 13 and find that we can achieve energy gains ranging between 3.7% and 11% on average, depending on the level of utilization. The reason why energy savings increase with utilization level is that there are more energy saving opportunities for iterative optimization at high utilization levels, i.e., the more work done by the server, the more iterative optimization saves energy because it gets the work done faster.

5.4 Benchmark applications

We also evaluate IODC with the benchmark applications and its 1000 data sets, albeit we reuse the data sets in these experiments. As explained in Section 4, the motivation for these experiments is to address several questions which would have been too time-consuming to explore with the large number of unique data sets using the MapReduce and server applications. Relevant questions are: (1) How far is the strategy from the optimal performance of iterative optimization? (2) Is there value in a strategy that factors in the costs and benefits of multiple programs at the same time? (3) How does the strategy perform when data sets are non-uniformly mixed? Because we can apply all 300 combinations on all data sets, it is possible to address all these issues for these benchmark applications. In this section, we essentially want to validate that the convergence profiles of the benchmark applications are similar to the profiles of the MapReduce and server applications, suggesting they respond similarly to IODC. We address points (2) and (3) in the next two sections.

As reported in Figure 14, the cumulated speedup curves are similar for the benchmark applications as for the MapReduce and server applications, i.e., performance speedup improves over time. The 'best' curve in Figure 14 corresponds to the performance obtained if an oracle would choose the best combination for each data set of each program, with all overheads (recompilations and training runs) factored in, though. In other words, it is the maximum performance that can be achieved, in theory. We find that the plateau performance of IODC is within 98% of this best possible performance, in large part because IODC can keep the overhead of recompilations and training runs small enough.

5.5 Shared budget

IODC manages the costs/savings budget for each program separately. Since multiple programs are usually run simultaneously within servers, one could also consider a *shared-budget* IODC strategy in which programs share their costs and savings. For instance, if one program has many good combinations, it will quickly achieve significant savings, and most of its latter explorations will be useless. On the other hand, a program with few good combinations may require a lot of time to converge, delaying potential savings. If these two programs were to share costs and savings, the second program could use some of the savings of the first program to find good combinations faster, increasing overall benefits.

We have setup an experiment to evaluate the shared-budget idea. `rsynth` is a program that converges slowly, even though it can potentially achieve a significant speedup; `susan_corner` converges quickly. When the programs do not share their budget, the convergence profiles are as shown in Figure 15(a). After a little while, `susan_corner` converges to a speedup of 1.5, while `rsynth`

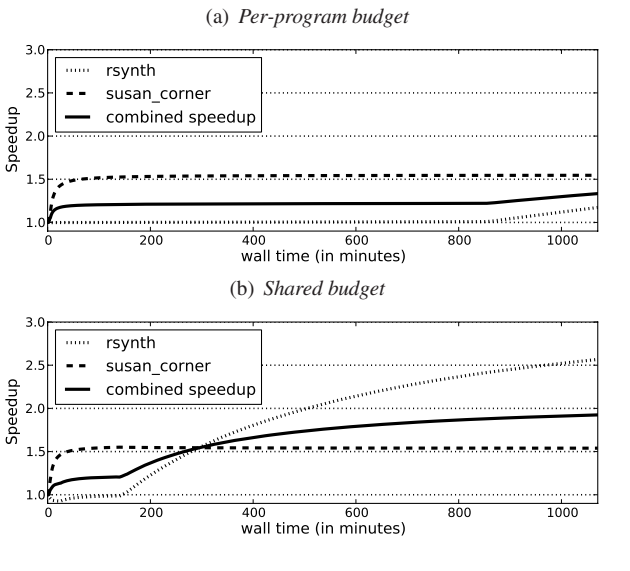

Figure 15. *Per-program vs. shared-budget IODC.*

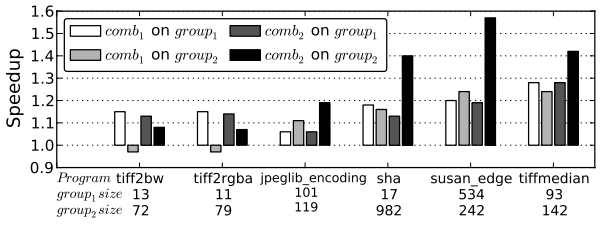

Figure 16. *Evaluating non-uniform data sets: Applying combinations, favored by one group of data sets, to another group of data sets.*

is still in the process of converging. The convergence profiles with a shared budget are shown in Figure 15(b): `rsynth` converges faster and achieves a higher speedup than with separate budgets in the same amount of time. `susan_corner`, which has contributed some of its savings to `rsynth`, performs only slightly worse than with separate budgets. Overall, the combined speedup is significantly higher within the same amount of time.

5.6 Non-uniform mix of data sets

In the experiments done so far, we have assumed the data set characteristics to be uniformly distributed over time. In the more likely case where each user runs data sets with certain common characteristics, data set characteristics are likely to be clustered over time. In this section, we emulate a scenario in which data set characteristics are clustered over time based on how they react to combinations. We cluster the data sets into two groups ($group_1$ and $group_2$) based on how they react to different combinations. We then find the best combination $comb_1$ based on the $group_1$ data sets (average performance across all $group_1$ data sets), and $comb_2$ only using $group_2$ data sets. While the spirit of this experiment is similar to the example of Section 2.4, here, we find $comb_1$ and $comb_2$ using a whole group of data sets, not just one, thereby emulating more closely the selection of combinations over multiple data sets, and the fact that data sets characteristics can be clustered over time.

In Figure 16, we present results for six programs. We apply $comb_1$ to $group_2$ and compare against the speedups obtained with $comb_2$ on $group_2$, and vice versa. For some benchmarks, $comb_1$ performs poorly for the $group_2$ data sets: similarly, $comb_2$ performs poorly for the $group_1$ data sets. This confirms the need for the continuous exploration of combinations of compiler optimizations, in case data set characteristics change over time. In Figures 17 and 18, we show that IODC can adapt to the changing data sets using `susan_edge` and `tiff2bw` as example benchmarks, respectively. The solid curves represent the default IODC strategy. The dotted curves show a variant in which the exploration stops after convergence. We start with data set $group_1$. After both strategies converged, we switch to data set $group_2$. As shown, with continuous exploration, IODC reacts quickly to the change, resulting in higher performance gains.

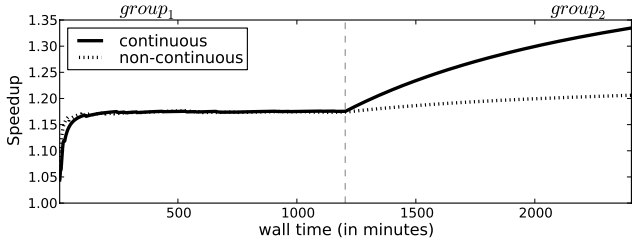

Figure 17. *Continuous vs. non-continuous exploration with non-uniform mix of data sets for `susan_edge`.*

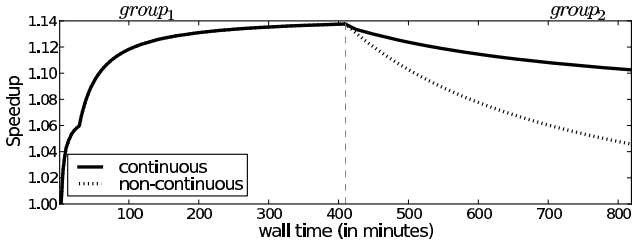

Figure 18. *Continuous vs. non-continuous exploration with non-uniform mix of data sets for `tiff2bw`.*

6. Related Work

MapReduce [11] has been applied to a broad set of algorithms since its introduction [10, 17, 21, 28, 35]. Aside from Google's proprietary implementation of MapReduce, other open-source implementations include Hadoop [33] and Sector/Sphere [18]. To the best of our knowledge, there is no prior work on applying iterative optimization to MapReduce, nor to the data center context at large.

However, there is a large body of research on iterative optimization. A large share of this research is dedicated to showing the performance potential of iterative optimization [1, 6, 8, 9, 13, 14, 25, 30]. A substantial number of studies aim at reducing the number of recompilations and training runs required to converge [1, 6, 8, 25, 30]. Some papers [15, 26] propose to improve the performance of iterative optimization by tuning at a finer granularity than the whole program. For most studies, the goal is to find the best possible combination, but not to find them in a cost-efficient manner that factors in the overhead of training, nor to adapt to the changing nature of data sets, i.e., it is assumed that iterative optimization can be done offline.

Most of the prior work in iterative optimization assumes a single or a limited number of data sets. A couple studies investigate input data set sensitivity. Berube and Amaral study the training input sensitivity of feedback-directed optimizations [4]. They collect 116 inputs in total for seven SPEC CPU benchmarks and study the impact of different training inputs on inlining decisions. Haneda et al. [19] investigate the input sensitivity of iterative optimization.

They perform iterative optimization using GCC and the `train` inputs on seven SPEC CPU benchmarks. The authors find that the best combination of optimizations found using the `train` input works well when applied to the `ref` input. Fursin et al. [14] proposed MiDataSets which includes 20 data sets in total per program. Only recently did Chen et al. [7] propose KDataSets, a suite of 1000 data sets per program, for evaluating iterative optimization with respect to data set sensitivity. They concluded that it is possible to find a program-optimal combination of compiler optimizations that performs well compared to the best possible combination for each specific data set. The program-optimal combination was chosen post-mortem though, with the assumption that all data sets are known, which we could not rely upon in the online system proposed in this paper.

A number of studies employ iterative optimization to optimize performance during run time. Voss et al. [36] propose ADAPT. They compile code sections into different versions and select the best performing ones during run-time in order to adapt to different architectures and inputs. Fursin et al. [16] use MiDataSets to understand how iterative optimization behaves in a more realistic setting where input data sets vary across executions. However, because of the limited number of input data sets, they have to reuse data sets, in contrast to this work. Their strategy is to generate a binary with two versions of the most time-consuming routines, each version optimized with a different combination. They then compare the performance of the two versions in a statistical way once enough calls to the two versions of the routines have been performed. They then pick the best performing version as the production combination. Stephenson [32] proposes a similar approach within a Java virtual machine. In contrast to this collection of prior work, IODC carefully manages the costs versus benefits, enabling online iterative optimization in the data center. In addition, IODC explores a larger combination space of compiler optimizations during run time.

7. Conclusion

Implementing iterative optimization in an efficient way so that it can be deployed in an online environment is non-trivial for a number of reasons: a potentially large number of recompilations and training runs may be needed to find a good combination of compiler optimizations; the process can be data set dependent; and the overhead of recompilations and training runs can easily wipe out the performance benefits from iterative optimization. In this paper, we proposed and explored Iterative Optimization for the Data Center (IODC), and we made the case that servers and data centers offer a context in which iterative optimization can be implemented as an online optimization technique. The key insight behind IODC is to carefully manage the number of recompilations and training runs so that they do not nullify the benefits from iterative optimization. We demonstrated that IODC can be applied to both MapReduce workloads as well as throughput compute-intensive server applications, and is completely transparent to the end user. We evaluated IODC using a very large number of data sets per benchmark, enough to use each data set (or record) only once. We report an average performance improvement of 1.48×, and up to 2.08×, for a set of MapReduce applications, and 1.14×, and up to 1.39×, for a set of contemporary throughput compute-intensive server applications.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. We also thank Xiaguang Qi, Wenting He, Dongni Han for helping collect some of the data sets used in this paper. Lieven Eeckhout is supported through the FWO projects G.0232.06, G.0255.08, and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement No. 259295. Olivier Temam is supported by HiPEAC-2 NoE under grant European FP7/ICT 217068 and INRIA YOUHUA associated team funding. The rest of the authors are supported by the National Natural Science Foundation of China under grants No. 60873057, 60921002 and 61033009; the National Basic Research Program of China under grant No. 2011CB302504; and the National Science and Technology Major Project of China under grants No. 2009ZX01036-001-002 and 2011ZX01028-001-002.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305, March 2006.

[2] N. Aghdaie and Y. Tamir. Implementation and evaluation of transparent fault-tolerant web service with kernel-level support. In *Proceedings of 11th International Conference on Computer Communications and Networks (ICCCN)*, pages 63–68. IEEE, October 2002.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in Large Databases. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, September 1994.

[4] P. Berube and J. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 251–260, March 2006.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[6] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, March 2007.

[7] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. W. u. Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)*, pages 448–459, June 2010.

[8] K. Cooper, P. 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, July 1999.

[9] 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 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 69–77, July 2005.

[10] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web (WWW)*, pages 271–280, May 2007.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, pages 107–113, December 2004.

[12] X. Fan, W. Weber, and L. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 13–23, 2007.

[13] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 78–86, July 2005.

[14] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High*

Performance Embedded Architectures & Compilers (HiPEAC), pages 245–260, January 2007.

[15] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, pages 29–46, November 2005.

[16] G. Fursin and O. Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, pages 34–49, January 2009.

[17] D. Gillick, A. Faria, and J. DeNero. MapReduce: Distributed computing for machine learning. Technical report, UC Berkeley, December 2006.

[18] Y. Gu and R. L. Grossman. Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. In *Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure*, June 2009.

[19] M. Haneda, P. Knijnenburg, and H. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 385–385, April 2006.

[20] 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, June 2004.

[21] Z. Liu, H. Li, and G. Miao. MapReduce-based backpropagation neural network over large scale mobile data. In *Proceedings of the 6th International Conference on Natural Computation (ICNC)*, August 2010.

[22] Loongson 2F. `http://www.loongson.cn/`.

[23] M. Marwah, S. Mishra, and C. Fetzer. Enhanced server fault-tolerance for improved user experience. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 167–176, June 2008.

[24] M. Nawaz, E. E. Enscore Jr, and I. Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *OMEGA: International Journal of Management Science*, pages 91–95, 1983.

[25] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, March 2006.

[26] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 173–181, September 2006.

[27] S. Patil and D. Lilja. Using Resampling Techniques to Compute Confidence Intervals for the Harmonic Mean of Rate-Based Performance Metrics. *Computer Architecture Letters*, pages 1–4, 2010.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, February 2007.

[29] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[30] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, March 2005.

[31] 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, June 2003.

[32] M. W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, MIT, USA, January 2006.

[33] Apache Hadoop. http://hadoop.apache.org, 2010.

[34] MovieLens data sets. http://www.grouplens.org/node/73, 2010.

[35] A. Verma, X. Llora, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. In *Proceedings of the 9th International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 13–18, November 2009.

[36] M. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 163–170, August 2000.