Practical Iterative Optimization for the Data Center

SHUANGDE FANG and WENWEN XU, SKLCA, ICT, CAS, China; Graduate School, CAS, China
YANG CHEN, Microsoft Research
LIEVEN EECKHOUT, Ghent University, Belgium
OLIVIER TEMAM, INRIA, Saclay, France
YUNJI CHEN, CHENGYONG WU and XIAOBING FENG, SKLCA, ICT, CAS, China

Iterative optimization is a simple but powerful approach that searches the best possible combination of compiler optimizations for a given workload. However, iterative optimization is plagued by several practical issues that prevent it from being widely used in practice: a large number of runs are required to find the best combination, the optimum combination is dataset dependent, and the exploration process incurs significant overhead that needs to be compensated for by performance benefits. Therefore, although iterative optimization has been shown to have a significant performance potential, it seldom is used in production compilers.

In this article, we propose iterative optimization for the data center (IODC): we show that the data center offers a context in which all of the preceding 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 it is transparent to the end user. To bring IODC to practice, we evaluate it in the presence of co-runners to better reflect real-life data center operation with multiple applications co-running per server. We enhance IODC

Extension of Conference Paper: This article is an extension of a previously published conference paper at ASPLOS 2012 [Chen et al. 2012]. The article contains the following contributions over the prior paper:

(1) We add experiments and analysis to understand whether iterative optimization and IODC are effective under the co-running scenario in Section 6.
(2) We propose and implement two practical techniques in Section 7 to enhance IODC with the capability of compatible scheduling and dynamic threshold adjustment, so as to guarantee improved robustness, and handle the extreme cases in the presence of performance noise due to co-running applications.

L. Eeckhout is supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259295. O. Temam is supported by a Google Faculty Research Award, the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), and the China 1000-talents program. Y. (Yungi) Chen is supported by the National Natural Science Foundation of China (NSFC) under grants 61100163, 61133004, 61222204, 61221062, 61303158, 61432016, 61472396, and 61473275; the National High Technology Research and Development Program of China under grant 2012AA012202; the Strategic Priority Research Program of CAS under grant XDA06010403; the International Collaboration Key Program of CAS under grant 171111KYSB20130002; and the China 10000-talents program. S. Fang, W. Xu, C. Wu, and X. Feng are supported by the National High Technology Research and Development Program of China under grant 2012AA010902; the NSFC under grants 60873057, 60921002, 60925009, 61033009, 61202055, and 61402445; and the National Basic Research Program of China under grant 2011CB302504.
David Whalley served as Editor-in-Chief for this submission.
Authors' addresses: S. Fang (corresponding author), W. Xu, Y. Chen, C. Wu, and X. Feng, Institute of Computing Technology, Chinese Academy of Sciences, No 6. Kexueyuan South Road, Haidian District, Beijing, China; emails: {fangshuangde, xuwenwen, cyj, cwu, fxb}@ict.ac.cn; Y. Chen, Microsoft Research, Beijing, China; email: chenyang.ict@gmail.com; L. Eeckhout, Department of Electronics and Information Systems, Ghent University, Belgium; email: lieven.eeckhout@elis.ugent.be; O. Temam, Inria Saclay, France; email: olivier.temam@inria.fr.

ACM 1544-3566/2015/05-ART15 $15.00
DOI: http://dx.doi.org/10.1145/2739048

with the capability to find compatible co-runners along with a mechanism to dynamically adjust the level of aggressiveness to improve its robustness in the presence of co-running applications.

We evaluate IODC using both MapReduce and compute-intensive throughput server applications. To reflect the large number of users interacting with the system, we gather a very large collection of datasets (up to hundreds of millions of unique datasets per program), for a total storage of 16.4TB and 850 days of CPU time. We report an average performance improvement of 1.48× and up to 2.08× for five MapReduce applications, and 1.12× and up to 1.39× for nine server applications. Furthermore, our experiments demonstrate that IODC is effective in the presence of co-runners, improving performance by greater than 13% compared to the worst possible co-runner schedule.

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

General Terms: Design, Performance

Additional Key Words and Phrases: Iterative optimization, compiler, MapReduce, server, data center, co-run

ACM Reference Format:
Shuangde Fang, Wenwen Xu, Yang Chen, Lieven Eeckhout, Olivier Temam, Yunji Chen, Chengyong Wu, and Xiaobing Feng. 2015. Practical iterative optimization for the data center. ACM Trans. Architec. Code Optim. 12, 2, Article 15 (May 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2739048

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, such as `-O3` in the GNU C compiler, corresponds to a preset 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 [Cooper et al. 2005; Franke et al. 2005; Kulkarni et al. 2004; Agakov et al. 2006; Stephenson et al. 2003; Fursin et al. 2007; Cooper et al. 1999; Cavazos et al. 2007], which is based on a simple but powerful concept: run a program multiple times, each time compiled with different combinations of compiler optimizations, to find the best combination for future use. Despite 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. First, 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. Second, the costly overhead of recompilation and training runs can easily wipe out the performance benefits of iterative optimization. Third, in most iterative optimization studies, the training runs are performed using the same dataset(s), even though in practice there is no point for a user to run the same data set multiple times.

In this article, 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 pptimization 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 by investing a fraction of the performance benefits of better performing combinations of compiler optimizations. The strategy operates automatically in the background and is transparent to the end

user—in other words, the user is unaware of the training runs and recompilations. We demonstrate the general applicability of IODC for both MapReduce-style as well as contemporary compute-intensive throughput server applications. We find the MapReduce framework [Dean and Ghemawat 2004] 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×. Then, we extend IODC toward non-MapReduce, compute-intensive throughput server applications. Considering nine benchmarks representative of compute-intensive throughput server applications, our strategy achieves an average speedup of 1.12×, and up to 1.39×.

There is one more challenge for iterative optimization in real-life data center environments: the interference of co-running programs, which we call *performance noise*. As applications are co-scheduled on the same server to improve hardware utilization [Mars et al. 2011; Kambadur et al. 2012], the performance variations induced by co-running applications could confuse IODC with regard to its effectiveness. Hence, as a second step in the article, we evaluate whether iterative optimization is still effective in the presence of co-runners. Our evaluation reveals that this is the case: iterative optimization can still find well-performing combinations of compiler optimizations despite performance noise; however, for some adversary co-runners, performance noise can be detrimental to the effectiveness of iterative optimization. Hence, we enhance IODC to be co-runner aware and co-schedule compatible applications along with a threshold-based mechanism to control the aggressiveness of iterative optimization to improve its overall robustness to performance noise. Our results show that the co-scheduling strategy improves performance by 13% compared to the worst possible co-schedule, whereas the threshold-based mechanism effectively dampens the aggressiveness of IODC in adversary co-runner scenarios.

Next to proposing and implementing IODC, we believe that this article is the first to create realistic conditions for the evaluation of iterative optimization by avoiding dataset reuse. We emulate data center operation by considering unique datasets across successive recompilations and training runs, which reflects many independent users interacting with the data center, all leading to different input datasets. In particular, for the MapReduce applications, we consider a very large number of unique datasets, varying between 1.55 million and 387.5 million unique records; for the server applications, we consider between 28,000 and 177,119 unique datasets, and we use each dataset only once throughout the exploration process.

In this article, we make the following contributions:

—We propose a novel online iterative optimization strategy for the data center (IODC) that balances overhead versus benefit, akin to 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 1.12×, and up to 1.39×, for a set of compute-intensive throughput server applications.
—We evaluate the effectiveness of iterative optimization in the presence of co-running applications per server—a likely scenario in the data center. We demonstrate that our threshold-based IODC strategy exhibits good robustness against performance noise for most co-runner scenarios and dampens the negative impact of adversary co-runners.
—We emulate realistic conditions in the experimental setup by considering a very large number of unique datasets, and we demonstrate that IODC can learn good

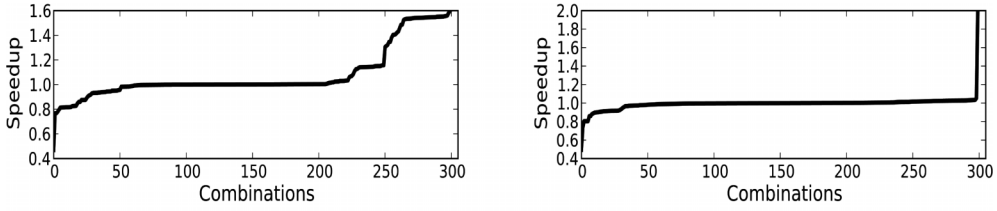

Fig. 1. Cumulative distribution of the average speedup for combinations of compiler optimizations ranked by increasing speedup for `susan_corner` (a) and `rsynth` (b), both from the MiBench benchmark suite [Guthaus et al. 2001].

combinations of compiler optimizations across unique datasets. We believe that this is the first article to do so in the iterative optimizations domain.

The remainder of this article is organized as follows. In Section 2, we illustrate the challenges of iterative optimization. In Section 3, we present the IODC strategy for both MapReduce and compute-intensive throughput server applications. We present the experimental setup in Section 4, followed by the evaluation of IODC in Section 5. In Section 6, we evaluate the sensitivity of IODC to performance noise. In Section 7, we enhance the IODC framework to mitigate noise-induced performance slowdowns. Finally, we discuss related work in Section 8 and conclude in Section 9.

2. MOTIVATION

We first introduce some terminology related to iterative optimization. When trying out a combination of compiler optimization, iterative optimization needs to compile the program with that combination, which we refer to as a *recompilation*, and run the compiled binary on a given dataset, 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 (GCC's `-O3`). We refer to the runs that produce actual user output as *productions runs*.

The potential of iterative optimization has been widely demonstrated [Cavazos et al. 2007; Fursin et al. 2007; Cooper et al. 2005]. However, it is plagued by several issues that prevent it from being broadly used in practice.

Iterative optimization requires many runs. Many combinations may need to be evaluated because of the complex optimization space. Figure 1 illustrates that the number of " good" combinations varies significantly across programs. (These exemplar results are obtained for a number of MiBench benchmarks [Guthaus et al. 2001] with KDataSets inputs [Chen et al. 2010] on an Intel Xeon platform—see Section 4 for more details about the experimental setup.) For each combination, we average the speedup over 1,000 datasets and rank the combinations by increasing average speedup. For `susan_corner`, more than 15% of the combinations yield a speedup of at least 5%. For `rsynth`, on the other hand, less than 0.4% of the combinations yield a large speedup. In other words, only a few combinations yield the best possible performance, and hence finding them may require a large number of recompilations and training runs.

Dataset sensitivity. It is challenging to find a combination that performs well on average across a broad set of datasets. We illustrate this point in Figure 2 for `tiff2bw` (also from MiBench [Guthaus et al. 2001]) and two datasets $dataset_1$ and $dataset_2$. For each of these two datasets, we determine the best combination among a set of 300 combinations (i.e., "$comb_1$ on $dataset_1$" and "$comb_2$ on $dataset_2$"). We subsequently apply each combination to the other dataset. The combination that performs best for one dataset may induce a slowdown for other datasets, as is the case for "$comb_1$ on $dataset_2$."

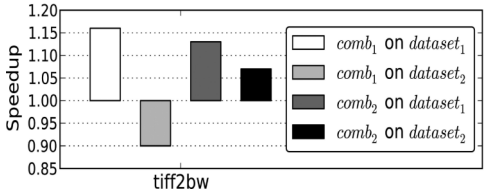

Fig. 2. Demonstrating dataset sensitivity for iterative optimization: applying combinations selected using a dataset to another dataset for `tiff2bw`.

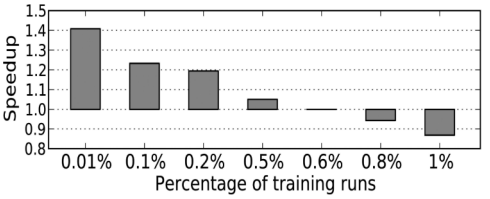

Fig. 3. Impact of the number of training runs on the overall performance of iterative optimization for `ann`.

In a real data center, it is to be expected that dataset characteristics change over time as users come and go and solve different problems. Hence, the best combination must be selected carefully, and the choice must be revisited constantly.

Overhead / benefit trade-off. Finally, the benefits of iterative optimization are not sufficient to tolerate the overhead of a massive number of recompilations and training runs; therefore, a careful trade-off between recompiling and training runs versus enjoying performance speedups from iterative optimization is required. In Figure 3, 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` (see Section 4). As little as 0.6% training runs are enough to wipe out any benefit from iterative optimization, and 1% of training runs result in a 13% slowdown over no iterative optimization. On the other hand, limiting the number of training runs to less than 0.1% yields substantial speedups.

3. ITERATIVE OPTIMIZATION FOR THE DATA CENTER

In this section, we first introduce IODC for MapReduce workloads, followed by IODC for compute-intensive throughput server workloads.

3.1. MapReduce Versus Iterative Optimization

MapReduce is a paradigm and framework for processing huge datasets on a distributed system [Dean and Ghemawat 2004]. MapReduce splits up computation in a Map and a Reduce stage. In the Map stage, the *master* partitions the input into smaller subproblems and distributes those to the *workers*. Each worker node works on a subproblem, which in itself consists of a number of *records*. In our setup, the entire problem consists of millions of records that are partitioned across the workers in subproblems, 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 seems to be a natural fit to how iterative optimization operates. Iterative optimization requires that a program be executed many times, albeit compiled with different combinations of compiler optimizations, to understand which combination of compiler optimizations yields optimum performance. In MapReduce, the master is running the iterative optimization strategy, whereas 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.

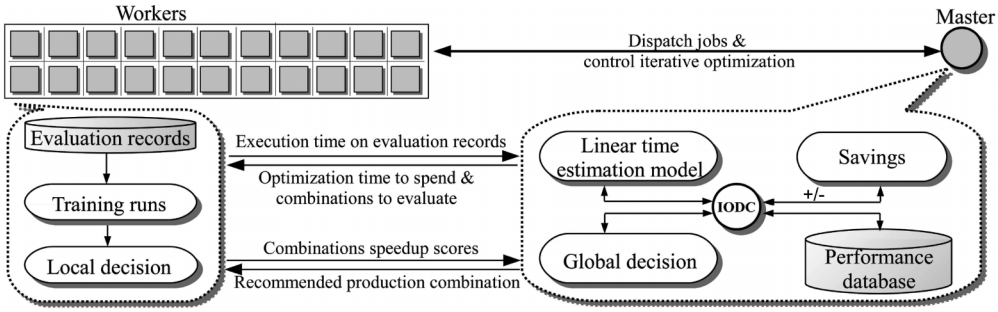

Fig. 4. IODC for MapReduce workloads.

3.2. IODC for MapReduce

Although the overall concept of IODC seems to be an intuitive fit for MapReduce, designing an efficient strategy is challenging. The main difficulty is making sure that the overhead of recompilations and training runs does not undermine the benefits of iterative optimization. The key principle of IODC is to use a novel algorithm for managing costs and benefits from iterative optimization carefully, much like a "savings account." IODC starts off with a zero balance in the savings account, and when the selected combination performs better than `-O3`, 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 in the savings account. Figure 4 illustrates the general flow of IODC for MapReduce workloads. The master dispatches the MapReduce jobs to the workers, controls iterative optimization, and keeps track of a performance database of previously evaluated combinations and the savings account. As mentioned before, evaluations of combinations are done on the fly, which is transparent to the user. 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 combinations in the background, hoping to identify a combination that yields even better performance. We 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. In addition, IODC will direct the workers to randomly select a sample of its assigned records and to use these as *evaluation records*. On all of these evaluation records, the workers run the Map and Reduce functions, compiled with the default compiler optimization (i.e., `-O3`). As well, 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 Figure 4). The evaluation records are also used to detect program errors caused by compiler bugs during recompilation; we will discuss this further in Section 3.4.

Estimating the performance gain by a combination. The strategy spends only a fraction of the savings brought by good combinations, 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 then 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 4.

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 to avoid recklessly spending the savings; we use P = 20% in our setup. The evolution of savings is illustrated in Figure 5. 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. In the worst case, such as if no better combinations are found, only a negligible loss is incurred.

Revisiting combinations versus evaluating new combinations. As explained earlier and shown in Figure 4, the master distributes to each worker a mix of new combinations and known combinations. The workers perform the explorative evaluations in parallel. The reason for looking for new combinations and revisiting older combinations is to be able to continuously find the best possible optimization, even in the presence of time-varying inputs. As soon as the evaluations are performed, the workers send the speedup scores for each evaluated combination to the master (in Figure 4, see the 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 workers (in Figure 4, see the bottom arrow from the master to the workers).

Within-run IODC versus 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—in other words, 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 Throughput Server Applications

Compute-intensive throughput 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 datasets 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 stand-alone script, running on one of the server nodes; the overhead of running the strategy script is factored in our evaluation, although it is actually negligible. As shown in Figure 7, the modifications to the strategy are the following.

Sample datasets. We maintain a pool of D randomly selected datasets, which we use the same way as the evaluation records for the MapReduce workloads: we use these evaluation datasets to evaluate the performance of alternative combinations of compiler optimizations. This pool is periodically refreshed as new datasets come in to adapt to the possibly changing nature of datasets 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). We use the t-test with resampling [Patil and Lilja 2010] to check whether the mean execution times (over the D datasets) are statistically significant. More specifically, we introduce a speedup threshold `T` (set to 1.02 by default). We only consider a combination for which the average speedup (on the selected samples) is greater than T as a candidate production combination. We then employ the t-test to check whether the speedup is statistically significant at the 95% confidence level. Once the t-test is passed, we replace the current production combination with the new combination.

Conservative and aggressive modes. Unlike the MapReduce workloads, 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 a priori. Therefore, exploration decisions have to be revisited after each run. The motivation for having two modes is that 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 willing 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 until the end of a mode are definitely won. Consider Figure 6. Initially, the strategy starts in aggressive mode, speculatively investing future savings to quickly find better combinations. In this example, we assume that the exploration performed on the first datasets finds some better-performing combination. The strategy will decide to stay in aggressive mode, postulating that even better combinations can be found. After several more combinations are evaluated without showing an additional benefit, the strategy toggles back to the 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 the aggressive mode and so on.

3.4. Implementing IODC

We now discuss several IODC implementation issues worth mentioning.

Compilation. For MapReduce IODC, we introduce a new API so that the user can specify the location of the source code of the Map and Reduce functions for automatic recompilation. Once the current best combination gets promoted as the next production combination, we send the combination itself to all MapReduce workers, which then perform recompilations locally. In the server implementation, IODC sends the recompiled code across the nodes. The time required for either propagation or recompilation is factored into our performance measurements.

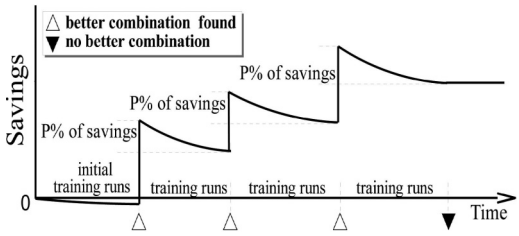

Fig. 5. Evolution of the IODC savings account.

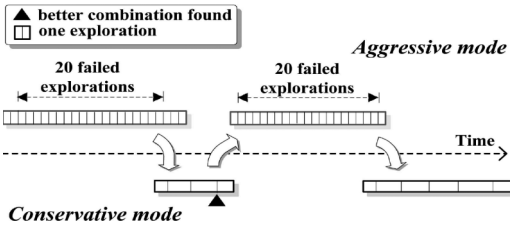

Fig. 6. Toggling between conservative and aggressive modes.

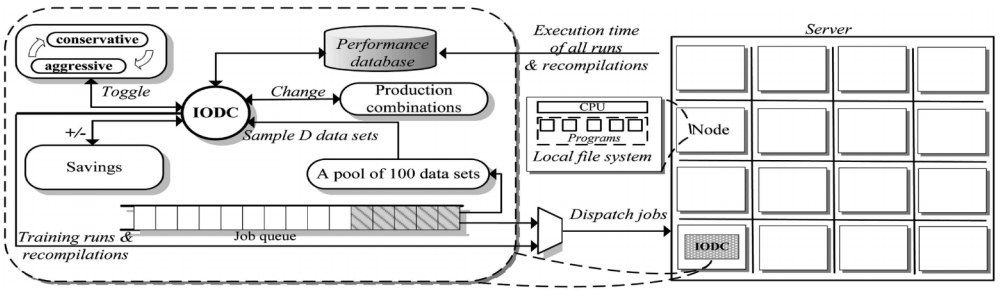

Fig. 7. IODC for compute-intensive throughput server applications.

Capturing inputs and outputs. To be transparent to the users, we need to automatically retain the inputs of evaluation runs and 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. Thus, we intercept messages between the master and the workers to know the input location and sample them for the evaluation runs. For retaining the *D* datasets for the server applications, we leverage the replay strategy that is commonly used in large-scale servers for fault-tolerant computing [Marwah et al. 2008; Aghdaie and Tamir 2002]—because of the high failure rate in servers, software infrastructures always embed a replay capability to retain the datasets and rerun the program if the previous execution failed.

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 add a safeguard mechanism to check the correctness of the compiled code. We leverage our evaluation records to evaluate correctness and the aforementioned runtime facilities for capturing the output. A new combination can be authorized for production runs only if it can produce the same output on the evaluation records as the default compiler optimization. Our experiments show that around 3% of the evaluated combinations generate bogus code; these combinations are then filtered out. The remaining combinations err in none or at most 0.02% of the subsequent runs (depending on the compiler).

4. EXPERIMENTAL SETUP

4.1. Servers

We evaluate IODC on three clusters: (1) a 7-node Dell *Intel cluster* with 3GHz Intel Xeon dual-core processors (E3110 family), 2GB RAM, 2x3 MB L2 cache, running CentOS release 5.3 (Final); (2) an *AMD cluster* with eight dual-core AMD Opteron processors (8218), 16GB RAM, 2x1 MB L2 cache, running Red Hat Enterprise Linux 4.4; and

(3) a 32-node *Loongsoncluster* with Loongson 2F processors [Loongson 2014], running Red Flag Linux 6.0—Loongson 2F is an 800MHz MIPS-compatible general-purpose CPU with a 512KB L2 cache and 512MB RAM [Loongson 2014]. To perform the very large number of runs described next in a reasonable amount of time, we partition our benchmarks across these three clusters as indicated in Table I.

4.2. Benchmarks and Datasets

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 [Das et al. 2007]. `ann` is a back-propagation algorithm for training the weights of a three-layer artificial neural network [Liu et al. 2010]; `kmeans` and `knn` are MapReduce implementations of k-means clustering [Ranger et al. 2007] and the k-nearest neighbor algorithm [Gillick et al. 2006], respectively; `pfsp` relies on a genetic algorithm and an NEH-based heuristic method [Nawaz et al. 1983] to solve the permutation flowshop scheduling problem, which is an important task in industrial engineering, aiming at finding a job sequence with minimal makespan [Verma et al. 2009].

The MapReduce implementation that we use is Sector/Sphere [Gu and Grossman 2009], a high-performance open-source distributed file system and parallel data processing engine. We generate 1,550,000 to 387,500,000 inputs (records) for these tasks, either based on a uniform distribution or a distribution extracted from real-world data (see Table I). We randomly group these records into five different datasets of the same size for each of the applications. We will use these five datasets to evaluate within-run IODC versus across-run IODC.

Compute-intensive server applications. The second group of workloads contains nine programs, eight of which are extracted from the PARSEC benchmark suite [Bienia et al. 2008], along with `bzip2e`, a well-known file compression utility taken from MiBench [Guthaus et al. 2001]. These programs are similar to emerging compute-intensive server applications used in several popular Web services (e.g., `x264` is the kind of tasks run on YouTube servers, and `freqmine` is a typical data mining program that Amazon could use similar algorithms to recommend related books). For these nine programs, we collect between 28,000 and 177,119 datasets, corresponding to a total storage of 14.7TB. In all experiments, we use each dataset only *once*, thereby emulating distinct production runs.

4.3. Compiler Optimization and Measurements

For all experiments, we use the GNU GCC v4.4 compiler. We measure end-to-end wall clock time and use the optimization level `-O3` as our baseline. We select 127 compiler options that control inlining, unrolling, vectorization, scheduling, register allocation, and constant propagation, among other optimizations known to have a potential impact on performance. All of these options are listed in Appendix A. We then create 300 randomly chosen combinations of these compiler optimizations. We explore these combinations in random order in the experiments. The performance numbers reported in the article include all sources of overhead, such as the cost of recompilations, training runs, and communication overhead between the master and the workers. The total CPU time was 110 days for the MapReduce applications and 740 days for the server applications (both co-run and solo-run experiments).

5. IODC PERFORMANCE EVALUATION

5.1. MapReduce Workloads

Figure 8 reports the speedup for 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

Table I. Benchmarks, Datasets, and Servers Considered in This Study

Benchmark Suite	Program (Domain)	Number of Datasets (Total File Size)	Dataset description	Cluster
MapReduce applications	minhash (probabilistic clustering)	10,830,935 (6GB)	Generated datasets obeying distribution extracted from MovieLens [MovieLens 2014]	Loongson
	ann (model training)	310,000,000 (222GB)	Generated datasets obeying uniform distribution	Loongson
	kmeans (clustering)	4,294,720 (320GB)	Generated datasets obeying uniform distribution; each record representing a multial point	Loongson
	knn (query-based learning)	387,500,000 (580GB)	Generated data sets obeying uniform distribution; each record consisting of an instance and its label	Loongson
	pfsp (scheduling problem)	1,550,000 (591MB)	Generated datasets obeying uniform distribution; each record representing a job scheduling scheme	Loongson
Server applications	bzip2e (compression)	28,000 (438GB)	Uncompressed tar balls of source and binary packages of various Linux distributions	Loongson
	freqmine (data mining)	61,655 (4.05TB)	Lists of transactions randomly generated using IBM Quest Market-Basket Synthetic Data Generator [Agarwal and Srikant 1994]	AMD
	x264 (video encoding)	67,571 (2.28TB)	Yuv4mepg videos converted from MKV and RMVB files downloaded from the Internet	Loongson
	blackscholes (financial analysis)	43,354 (434GB)	Lists of options randomly generated based on inputs provided by PARSEC	AMD
	ferret (image similarity search)	87,720 (263GB)	Groups of query images constructed using 1,166,657 JPEG files downloaded from 130 different Web sites	Intel
	canneal (electronic design automation)	177,119 (2.15TB)	Netlists randomly generated with an enhanced version of the input generator provided by PARSEC	Intel
	vips (image processing)	56,000 (2.96TB)	Images downloaded from 130 different web sites.	Intel
	streamcluster (stream clustering)	70,000 (1.20TB)	Randomly generated multidimensional points	Intel
	kmeans (clustering)	65,000 (936GB)	Generated datasets obeying uniform distribution; each contains a group of multidimensional points	Intel

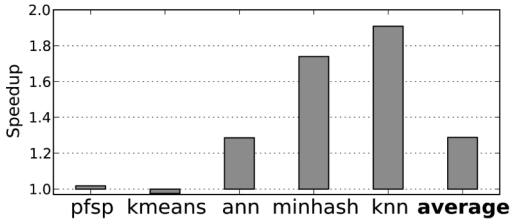

Fig. 8. Speedup for the MapReduce applications for within-run IODC.

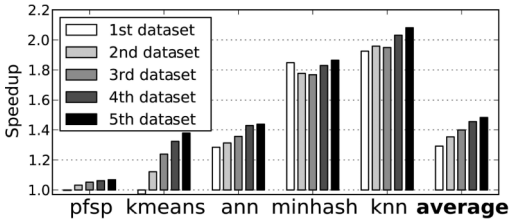

Fig. 9. Speedup for the MapReduce applications using both within-run IODC (first dataset) and across-run IODC (second through fifth datasets).

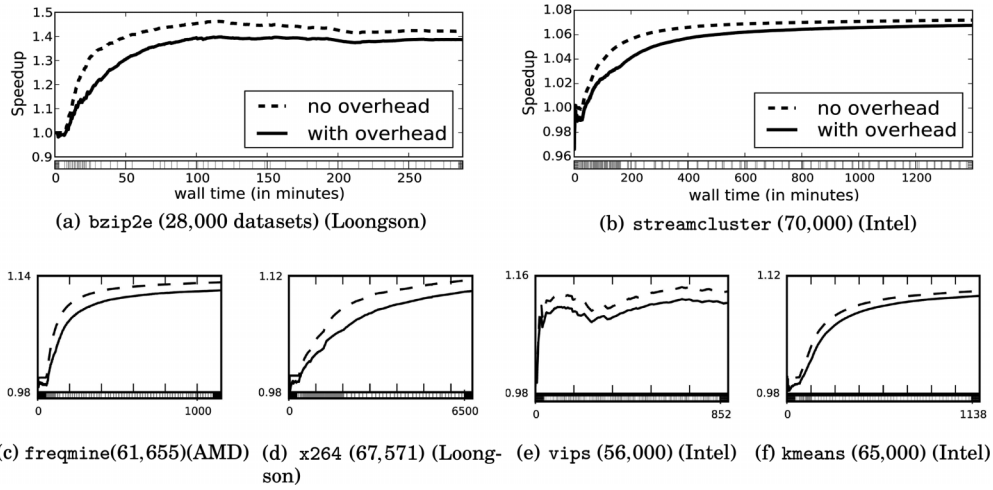

Fig. 10. Cumulated speedup for `bzip2e` and `blackscholes`.

to the initial "investment" in the exploration of combinations that failed to find a better combination over GCC's default `-O3` optimization level and which results in a 1.24% slowdown. Note though that we perform within-run IODC only in this experiment.

To evaluate across-run IODC, we consider five (distinct) datasets to emulate five consecutive runs of the same MapReduce workload with different inputs (Figure 9). On average, the speedup achieved across runs equals 1.48× after five datasets versus 1.29× after one dataset. The maximum speedup is 2.08×, achieved for `knn` after five datasets. Note that we now achieve a speedup of 1.38× after five datasets of across-run IODC for `kmeans`, which showed a 1.24% slowdown through within-run IODC.

5.2. Server Workloads

In Figure 10, we report the cumulated speedup for the server applications using unique datasets. 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 the production combinations. The "with overhead" curve accounts for all overhead costs, whereas the "no overhead" curve discounts any source of overhead; the delta between both curves quantifies the overhead.

Table II. Production Combination Found by IODC for a Number of Example Benchmarks

Benchmark	Combination
ann	-O3 -fira-share-spill-slots -fno-unswitch-loops -freorder-blocks-and-partition -funroll-all-loops
pfsp	-O2 -falign-functions -falign-functions=28 -finline-small-functions -fipa-pure-const -fno-branch-count-reg -fno-if-conversion -fno-schedule-insns -fno-tree-dominator-opts -fno-tree-sink -fno-tree-sra -fsched-spec-load -fthread-jumps -funroll-all-loops
blackscholes	-Os –param max-inline-insns-recursive=181 -fcrossjumping -fdse -fgcse-lm -fmodulo-sched -fno-align-functions -fno-defer-pop -fno-function-cse -fno-schedule-insns -fno-web -fpeel-loops -fsched-spec -ftree-dse
bzip2	-Os –param max-unrolled-insns=2028 -ffunction-cse -floop-interchange -fno-ivopts -fno-tree-loop-ivcanon -fno-tree-sink -fno-tree-vect-loop-version -fno-vect-cost-model -fsched-stalled-insns=34 -fsee -ftree-ccp -ftree-reassoc
canneal	-O3 –param max-inline-insns-recursive-auto=444 -falign-functions -fbranch-count-reg -fbtr-bb-exclusive -fif-conversion2 -fipa-cp-clone -fno-expensive-optimizations -fno-inline-functions-called-once -fno-loop-interchange -fno-peephole -fno-reorder-blocks-and-partition -fno-tree-ter -fno-vect-cost-model -fschedule-insns

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 (2,670 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. We show the cumulated speedup profiles for six out of nine benchmarks (including three newly added benchmarks over the conference paper) in Figure 10; the other three benchmarks are shown in the conference paper [Chen et al. 2012]. Overall, the proposed strategy achieves a cumulated speedup of 1.39× for `bzip2e`, 1.12× for `blackscholes` and `freqmine`, 1.10× for `x264`, 1.09× for `kmeans` and `vips`, 1.08× for `canneal`, 1.07× for `streamcluster`, and 1.06× for `ferret`, with an average of 1.12×.

Additionally, note the thin marks at the bottom of each of the graphs in Figure 10. 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, then after the inflection point it starts increasing faster. The aggressive mode is reflected right before the inflection point. Once a better combination is found, the strategy transitions back to the conservative mode, after which performance benefit increases again.

As concrete examples, Table II lists, for some of the programs, the best production combination found by IODC during the experiments. Due to space constraints, we omit the combinations found for other programs, as they contain too many (up to more than 100) options. Overall, these experiments show that even under conditions with unique datasets and factoring in all overheads, IODC improves performance in a consistent way—without incurring performance degradation at any time except in the initial phase.

5.3. Discussion

In the preceding experiments, we use random search to explore the astronomically large compiler optimization space. We find that hundreds of explorations and in some cases fewer explorations suffice to find combinations outperforming the most aggressive default optimization level (e.g., GCC's `-O3`). Of course, the exact numbers and the resulting speedups vary significantly across programs, platforms, and compilers. This observation is consistent with previous studies [Agakov et al. 2006; Hoste and Eeckhout 2008; Chen et al. 2010]. Machine learning–based and other more sophisticated

algorithms [Kulkarni et al. 2004; Agakov et al. 2006] have been proposed to accelerate the search process. Many of these need a bootstrap or training phase that requires random search. As the search algorithm is orthogonal to other components of IODC, it can be easily replaced by or augmented with these algorithms.

We now discuss the following questions. (1) How far off 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 datasets are nonuniformly mixed? We briefly answer them here, and more details can be found in the original conference paper [Chen et al. 2012]. For the first question, we find that steady-state performance of IODC is within 98% of the best possible performance, in large part because IODC can keep the overhead of recompilations and training runs small. For the second question, the answer is positive: the results show that if two programs were to share costs and savings, one program could use some of the savings of the other program to find good combinations faster, increasing overall benefits. Especially in a case in which one program has many good combinations, it will quickly achieve significant savings, and most of its explorations will be useless. Finally, in the more likely case in which each user runs datasets with certain common characteristics, dataset characteristics are likely to be clustered over time. With online continuous exploration, IODC can quickly react to the change in dataset characteristics, resulting in higher performance gains.

6. EVALUATING IODC IN THE PRESENCE OF PERFORMANCE NOISE

The interference of co-running applications is common in a data center, because applications are usually scheduled on shared machines to improve hardware utilization. Previous studies [Mars et al. 2011; Kambadur et al. 2012] reveal that interference is incurred by the complicated interactions of co-running programs. They show that co-runners can have a significant performance impact in a real data center production environment—a phenomenon we refer to as *performance noise*. Since iterative optimization and IODC are based on the careful performance evaluation of combinations of compiler optimizations, it is likely vulnerable to the interference caused by co-runners. In this section, we first investigate some fundamental issues of iterative optimization itself under a co-runner scenario. This will help us better understand whether iterative optimization is still effective in the presence of co-runners; our results show that IODC is inherently robust to performance noise. Finally, we introduce two approaches to further enhance IODC's robustness.

Methodology. To study the impact of performance noise on the effectiveness of iterative optimization, we consider all pairs of the five server benchmarks (`ferret`, `canneal`, `vips`, `streamcluster`, and `kmeans`) that run on our Intel cluster—20 co-runner workloads in total. We use unique datasets in the experiments, or in other words, a benchmark runs over and over again, each time given a new input data set, to mimic many independent, subsequent jobs in the data center. When co-running benchmarks, we stop the experiment as soon as one of the benchmark runs out of its input datasets. For the rest of the experiments in the article, we use a newer version of the GNU C compiler (v4.7), and we consider 192 compiler options. We also follow the random approach mentioned before for creating and exploring the combinations of compiler options.

6.1. Iterative Optimization under Performance Noise

In this section, we study the impact of co-runner interference on iterative optimization. There are two fundamental questions we want to ask and understand: (1) do good combinations of compiler optimizations exist in presence of interfering co-runners? and (2), if so, are the good combinations different for different co-runners?

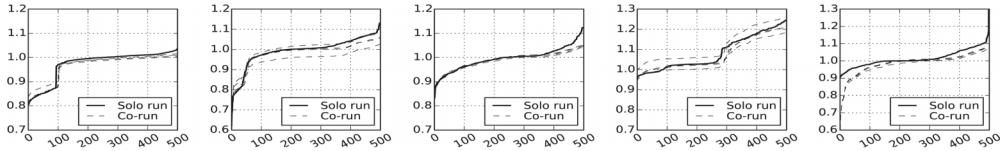

Fig. 11. Speedup obtained through iterative optimization as a function of the combinations (sorted) for isolated execution (solid line) and co-run executions (dashed lines).

To explore these two issues, we select 192 GCC compiler optimizations that are known to have a potential impact on performance. We then randomly compose these compiler optimizations to form 1,000 combinations of optimizations. We run all of these 1,000 combinations on all datasets for each program, and we retain the 500 combinations with the best average performance. We rank these combinations by increasing average performance (considering `-O3` as our baseline) and show the ranked speedup distribution of combinations as solid curves in Figure 11, assuming an isolated run (`solo_run`). We then repeat the preceding experiment for each co-run program pair to reveal the ranked speedup distribution of combinations under interference of co-runners and show its performance using dashed curves. Note that all curves are drawn considering their own combination ranking—that is, the same x-axis may indicate different combinations for the different curves. The answer to the first question is clear from this result—for each co-run program pair, there exist good combinations (i.e., better than `-O3`) in the presence of interfering co-runners. The results also show that the performance distribution of combinations exhibit a similar shape in both the isolated and co-run scenarios. In other words, iterative optimization improves program performance even in the presence of co-run interference.

However, the answer to the second question (if the good combinations are different for different co-runners) is no. In our experiments, for each program, we consider a co-runner scenario with each of the other four programs. We select the top-20 combinations out of the 500 combinations for each case and report the ratio of common ones that are shared among all four co-runner scenarios (Table III). We also indicate whether all co-runner scenarios share the best combination in the last column. The results show that for `canneal`, `streamcluster`, and `vips`, the best combinations under different co-running scenarios are different. In most cases, good combinations are not identical among different co-run scenarios, especially for `canneal` and `streamcluster`. However, for `ferret`, `vips`, and `kmeans`, more than half of the good combinations are the same for different co-runner scenarios—that is, these programs are insensitive to different co-runners. According to previous research [Tang et al. 2011; Kambadur et al. 2012], programs are much more vulnerable to the co-runners if they are sensitive to a shared resource such as the last-level cache. In Figure 12. we re[prt the L2 cache access ratio (L2 cache accesses per 1,000 CPU cycles) of each benchmark running unique input datasets. The data is collected using PAPI [PAPI 5.1 2013] during isolated executions. The results are in line with the observations mentioned earlier: `ferret`, `vips`, and `kmeans` require much less L2 cache resources; on the other hand, `streamcluster` induces much higher memory traffic than the other programs, and the access behavior of `canneal` is very unstable, which makes it sensitive to its co-runners.

6.2. IODC under Performance Noise

Considering that iterative optimization is still effective under a co-run scenario, as discussed in the previous section, we need to find different combinations when a program is co-run with different co-runners using IODC. Recall that the key idea of the IODC

Table III. Shared Combinations

Program	Ratio of Shared Combinations	Share Best (Yes/No)
canneal	0.45 (9/20)	No
streamcluster	0.40 (8/20)	No
ferret	0.80 (16/20)	Yes
kmeans	0.75 (15/20)	Yes
vips	0.55 (11/20)	No

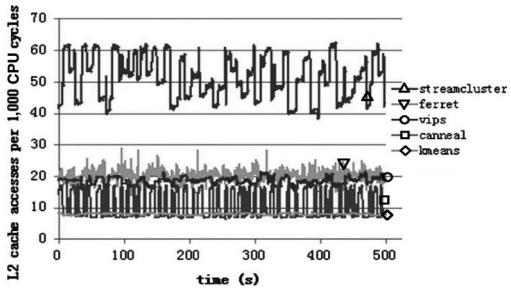

Fig. 12. L2 cache accesses per 1,000 CPU cycles.

strategy is to carefully manage the costs and benefits of the exploration of compiler optimizations. Interferences (performance noise) in the co-running scenario may pose challenges to the estimation of the costs and benefits. Unreliable estimations may lead to no benefit and performance slowdowns in the case of adversary co-runners. Thus, in this section, we first formally analyze the robustness of IODC under performance noise; we then experimentally evaluate the IODC strategy under the co-run scenario.

Analysis of IODC under performance noise. The benefit brought by the production combination is estimated using Equation (1). S_{pdt} is the speedup of the production combination in current use. This combination was evaluated and chosen in one of the past IODC explorations, and its speedup is estimated on a sample dataset. *pdt_time* is the accumulated production execution time after switching to this production combination. Hence, if we overestimate S_{pdt} because of the interference of co-runners, IODC will spend too much time exploring new combinations. However, if our strategy only invests $P\%$ of the benefits to do the explorations, there should be some robustness against the inaccurate estimation. Suppose that the actual speedup of a production combination equals S_{acc}; the actual benefit after doing the explorations using $P = 50\%$ of the overestimated benefits is shown in Equation (2).

$$Benefit = (S_{pdt} - 1) \times pdt\_time \tag{1}$$

$$Benefit_{left} = (S_{acc} - 1) \times pdt\_time - P \times (S_{pdt} - 1) \times pdt\_time, \quad with\ P = 50\% \tag{2}$$

$$y = f(x) = 2 - \frac{1}{x}, \quad with\ x = S_{acc}\ and\ y = \frac{S_{pdt}}{S_{acc}} \tag{3}$$

As we want to further investigate how much overestimation of speedup of the production combination we can tolerate, we draw the critical curve—that is, $Benefit_{left} == 0$ (Equation (3)), the solid curve with the $\times$ markers in Figure 13. We show the curve in the form of overestimation ratio (i.e., $\frac{S_{pdt}}{S_{acc}}$ on the vertical axis) varying along with every possible S_{acc} value (i.e., the horizontal axis). This curve actually represents the function in Equation (3). According to the geometric meaning of this curve, it divides the plane into two parts: the bottom right one indicates the $Benefit_{left} > 0$, and the upper left one means $Benefit_{left} < 0$. Besides, we only consider the case that $S_{pdt} > S_{acc}$, which indicates that the speedup of the production combination is overestimated (Equation (4)), and the area above the solid horizontal line in Figure 13. In addition, we only consider the overestimation under $2\times$, which is the most likely case—that is, the area below the

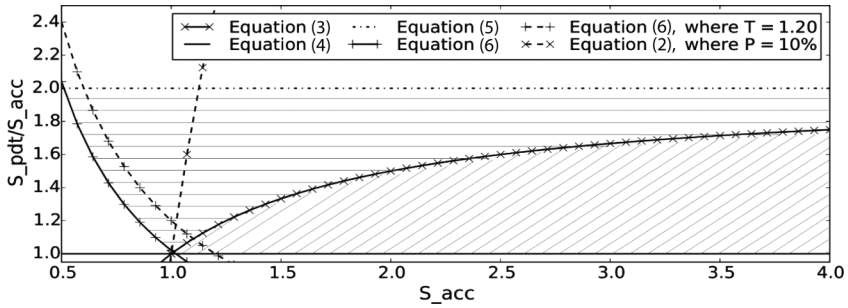

Fig. 13. Theoretical analysis of IODC under performance noise.

dot dash horizontal line in Figure 13. The condition is shown in Equation (5).

$$y = g_1(x) >= 1, \quad where\ y = \frac{S_{pdt}}{S_{acc}} \tag{4}$$

$$y = g_2(x) <= 2, \quad where\ y = \frac{S_{pdt}}{S_{acc}} \tag{5}$$

$$y = z(x) >= \frac{T}{x}, \quad where\ x = S_{acc}, y = \frac{S_{pdt}}{S_{acc}}\ and\ T = 1.02 \tag{6}$$

Besides, in the IODC strategy, we use a threshold-based mechanism to filter out measurement noise. One of the thresholds (defined as T) is to judge whether a new combination is significantly better than the current production combination. We set T to 1.02 by default, which means that only the combinations whose speedup is greater than 1.02 will be considered as candidates for the new production combination. Therefore, we draw the solid curve with a + marker, which represents the threshold in Figure 13. We only consider the right side of this curve, as only the combinations accepted as production ones can have impact on the overall benefit. The constraint inequalities are shown in Equation (6). Consequently, in the figure, the area filled with diagonal and horizontal lines satisfies all constraints of Equations (4), (5), and (6). Whereas Equation (3) divides this area into two parts (as mentioned earlier), the bottom right area (filled with diagonal lines) indicates the cases that IODC can handle (i.e., $Benefit_{left} > 0$), even if the speedup is overestimated; and upper left area (filled with horizontal lines) shows the cases in which IODC may cause slowdown. We can draw two conclusions from Figure 13. First, IODC can handle a large part of the overestimated cases, which can provide the strategy decent robustness against the interference in the co-run data center environments. Most overestimated cases can be handled if we dynamically adjust the thresholds of IODC, as we will discuss in more detail in Section 7. Second, the smaller the speedup of the production combination (i.e., S_{acc}), the more sensitive it is to performance noise. The extreme case occurs in case S_{acc} equals 1.0 (i.e., the same as `-O3`), in which no overestimation can be tolerated; otherwise, IODC will incur a performance slowdown.

Evaluation of IODC under a co-run scenario. We extensively evaluate IODC under the co-run scenario to understand whether the inherent robustness (as discussed earlier) is enough to preserve the overall benefit of iterative optimization. We apply IODC to each benchmark executed together with a co-runner on another core of the same cluster node. We compute speedup by considering wall clock time of a combination relative to `-O3` for the same co-running scenario: $\frac{\sum_{d \in datasets} time_{O3}^{corun}(d)}{\sum_{d \in datasets} time^{corun}(d)}$. In Figure 14, we observe that `ferret`

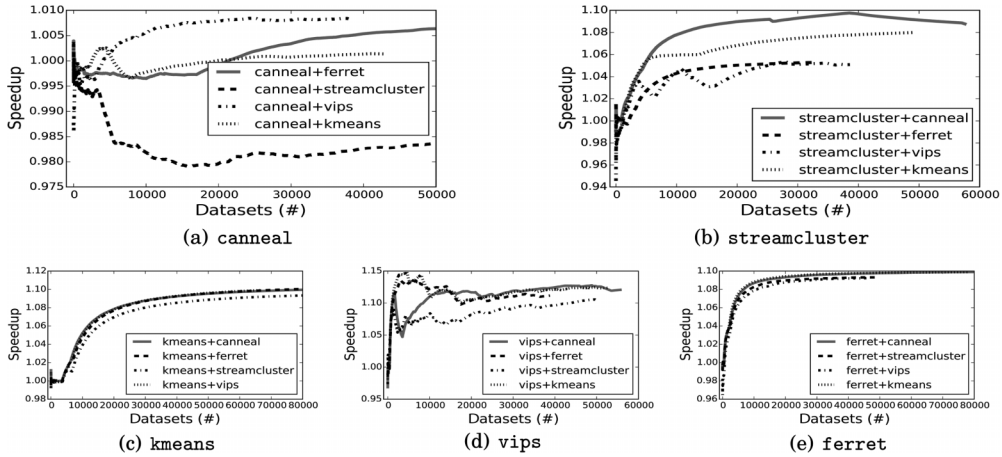

Fig. 14. Cumulated speedup profiles with IODC over co-run with `-O3`.

Table IV. Statistics Regarding Optimization Trials and Changes: Number of Times the Combination of Optimizations Is Changed, the Number of Trial Combinations, and the Number of Datasets Used during the Exploration

Program + Co-runner	Isolated Run, Changes/Trials (Datasets)	Co-run
canneal+streamcluster	1/39 (50,000)	49/140 (50,000)
canneal+ferret		1/51 (50,000)
canneal+vips		1/34 (38,399)
canneal+kmeans		3/53 (43,147)
streamcluster+canneal	4/86 (70,000)	12/155 (63,523)
streamcluster+ferret		3/46 (33,096)
streamcluster+vips		23/124 (38,756)
streamcluster+kmeans		5/117 (49,286)
ferret+canneal	2/63 (50,000)	2/84 (87,720)
ferret+streamcluster		2/46 (50,000)
ferret+vips		2/56 (48,125)
ferret+kmeans		2/81 (79,569)
vips+canneal	209/429 (50,000)	185/424 (56,000)
vips+streamcluster		181/371 (50,000)
vips+ferret		101/280 (39,000)
vips+kmeans		149/393 (53,258)
kmeans+canneal	4/99 (80,001)	3/92 (80,001)
kmeans+streamcluster		4/90 (80,001)
kmeans+ferret		2/89 (79,001)
kmeans+vips		3/75 (66,501)

and `kmeans` are largely insensitive to co-runners, and IODC achieves almost the same steady-state performance in the presence of co-runners as under isolated execution—all three co-runners have negligible impact on performance. On the other hand, co-runners have a significant impact on `streamcluster`, `vips`, and `canneal`; however, in most cases (19 out of 20), IODC can tolerate the interference (i.e., the cases in the diagonal area in Figure 13) and finds good combinations of compiler optimizations to accumulate performance benefits over time. The only exception is `canneal` when co-run with `streamcluster`, which indicates a small but consistent slowdown of approximately 3%.

To better understand the impact of co-runners on IODC, we refer to Table IV, which reports the number of combinations considered along with the number of times a

new (presumably better) combination is selected during both the isolated and co-run scenarios. (Recall that a combination is evaluated once enough time benefits have been accumulated; a combination is selected if it proves to outperform the previous best combination.) These results explain what is happening when `canneal` is co-run with `streamcluster`. In the presence of `streamcluster`, `canneal` makes 49 combination changes instead of 1 in the isolated run (and evaluates 140 combinations instead of 39). Due to performance noise induced by `streamcluster` and the small performance improvements that `canneal` achieves through iterative optimization (smallest among all five benchmarks (see Figure 11)), a new combination is frequently, and incorrectly, believed to outperform the previous best combination and thus is selected. As a result, the strategy keeps trying out new combinations, because of the overestimated benefits, but in fact it keeps accumulating losses, leading to a slowdown. Essentially, as good combinations are rare for `canneal` the speedup achieved through iterative optimization (i.e., S_{acc}) is very small. In other words, most of overestimated points occur around $S_{acc} == 1$ (see Figure 13): in this area, the costs of iterative optimization may wipe out any benefits. To address these idiosyncratic cases, we enhance IODC to dynamically adjust the aggressiveness of iterative optimization to prevent IODC from yielding significant performance losses in such cases.

7. MITIGATING NOISE-INDUCED SLOWDOWN

To mitigate noise-induced slowdown in IODC, we learn the *compatibility* among potential co-runners and then select the most compatible programs to be co-scheduled. The spirit of our approach is to try out co-run schedules at the beginning, on which we can draw some experiences to guide the scheduling for subsequent runs. This is in line with the spirit that a few programs are run a large number of times in a data center context. More specifically, we augment IODC with the ability to monitor the degree of mutual influence (DMI) between programs. We maintain a DMI table during the lifetime of the programs. We use this table to guide the scheduling. The DMI is computed as follows. Consider, without loss of generality, two co-running programs P_1 and P_2; over many datasets, we consider both of them running with the time span of T (albeit a different number of datasets each). We call $S(P_1, P_2)$ the speedup (slowdown in fact, i.e., <1) of P_1 co-run with P_2, which is calculated as $S(P_1, P_2) = \frac{T_{P_1\_O_3\_solo}}{T}$, with $T_{P_1\_O_3\_solo}$ the isolated execution time of P_1 using the baseline combination (`-O3`). Thus, the average speedup of the pair with respect to the sequential solo runs equals $DMI(P_1, P_2) = \frac{T_{P_1\_O_3\_solo} + T_{P_2\_O_3\_solo}}{2 \times T} = \frac{(S(P_1,P_2)+S(P_2,P_1)) \times T}{2 \times T} = \frac{(S(P_1,P_2)+S(P_2,P_1))}{2}$. Note that $T_{P_i\_O_3\_solo}$ and T are easily measured/estimated during execution. DMI thus captures how much the two programs influence each other. The DMI for a group of more than two co-running programs can be defined similarly. A low DMI indicates incompatible programs, whereas a high DMI indicates compatible programs. We select the co-schedule that minimizes the overall DMI value (i.e., the sum of the DMIs of all co-running program groups). In practice, it might not be always feasible to evaluate all possible co-schedules and find the one with a global minimum cost, because some of the DMIs might not have been measured or because there are too many candidate co-schedules. As a result, a co-schedule with local minimum cost across all of the already evaluated co-schedules is typically considered. This decision needs to be revisited whenever new DMI values are measured and, of course, whenever programs running in the system change as jobs come and go. Studying more efficient ways to gradually explore the co-schedule space is beyond the scope of this article and is left for future work.

We demonstrate this scheduling strategy in Figure 15 by reporting the average cumulated speedup (slowdown in fact, as solo run is baseline) profiles of each scheduling scheme (i.e., averaged on four programs for each curve). The scheme `canneal/vips`

+ `streamcluster/ferret` with the biggest total DMI (i.e., 1.84) achieves the highest cumulated speedup, which is 13% better over the worst scheme.

Adjusting IODC's aggressiveness. Even though we have shown how to properly co-schedule programs to improve IODC's robustness, it may still happen that two incompatible co-runners have to be scheduled together. Hence, we still need to mitigate the negative impact of "misbehaving" co-runners to the extent possible. The key to our solution is to leverage the parameters in the IODC strategy to decrease its aggressiveness. As discussed previously, parameters P and T are critical to IODC and affect its aggressiveness. Recall that we set $P = 50\%$ by default in the aggressive mode; decreasing P reduces the amount of savings invested to explore new combinations, thereby reducing the risk of overspending. This is illustrated in Figure 13, which shows the critical benefit curve (i.e., $benefit_{left} == 0$) using $P = 10\%$; see the dashed curve with $\times$ marker. Compared to the solid curve, the dashed one moves up so that more overestimated cases can be handled by IODC. Another option is to raise the threshold T at which a new combination is deemed better than the current combination. As shown in Figure 13, the dashed curve with a $+$ marker represents Equation (6) with T set to 1.20. Clearly, it is a two-edged sword, as both failing (horizontal line area) and successful (diagonal line area) cases of IODC will be pruned. In other words, if the threshold T is high, performance noise is less likely to skew the performance comparison of a trial combination against the current combination. However, if the threshold is set too high, it will also become difficult to accept a new and potentially better combination.

In essence, the threshold should only be raised if there are serious interferences causing severe performance slowdowns. However, it is difficult to judge the status of iterative optimization using just the speedup of the current production combination. Yet we can easily detect whether a co-run pair performs poorly using the cumulated speedup estimation, as illustrated in Figure 14. The problem is that we cannot get the real cumulated speedup profile in a production environment, as this would require running all datasets using both the production and baseline combinations. To circumvent this issue, we use history information of the production and baseline combinations on the training datasets to estimate the cumulated speedup of the program. We first estimate the cumulated speedup without overhead of iterative optimization using Equation (7); $time_{O3}(d)$ and $time_p(d)$ are the execution times of `-O3` and production combination on training dataset d, respectively. We then also need to factor in the cost of the trial executions (recompilation and execution) that were conducted during the tenure of combination c (i.e., the overhead) to get a realistic estimated of the cumulated speedup CS_{real} (Equation (8)):

$$CS_c = \frac{\sum_{d \in \text{training datasets}} time_{O3}(d)}{\sum_{d \in \text{training datasets}} time_p(d)} \tag{7}$$

$$CS_{real} = \frac{\sum_{d \in datasets} time_c(d)}{\sum_{d \in datasets} time_c(d) + \sum_{i=0}^{opt\_trials} overhead(i)} \times CS_c, \tag{8}$$

where *datasets* represents all datasets that have been run, and $overhead(i)$ represents the cost of the ith iterative optimization trial, including re-compilation and evaluation.

We add an additional check on the CS_{real} before acting any combination change; the threshold is deemed too low (susceptible to noise) whenever CS_{real} is less than 1. In this case, threshold T is increased by 0.01. Once we have detected that the threshold

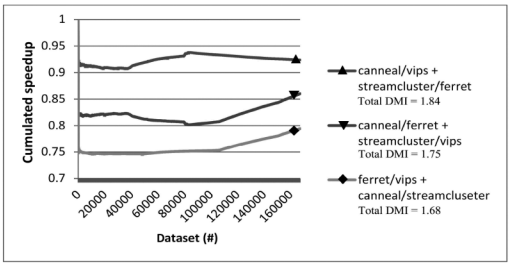

Fig. 15. Co-scheduling alternatives for four programs on two cores; the relative order of the curves is consistent with their respective DMIs.

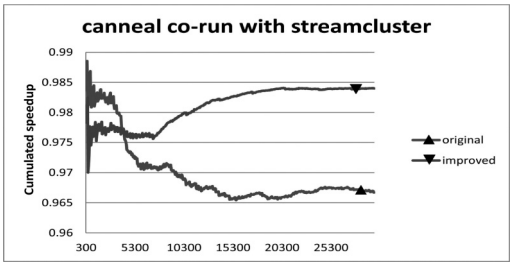

Fig. 16. Cumulated speedup with (and without) IODC threshold adjustment.

is too low, we also reduce P (i.e., we reduce the amount of savings consumed to evaluate new combinations). In Figure 16, we show the estimated cumulated speedup before and after applying this modified iterative optimization strategy for the worst co-runner pairs (i.e., `canneal + streamcluster`); the baseline is *co-run*`-O3`. (The other benchmark pairs were not affected significantly and are omitted here because of space constraints.) We can see that the cumulated speedup profiles are sufficient to decide whether the iterative optimization strategy is having a positive or negative effect and adjust accordingly to avoid a significant performance slowdown.

8. RELATED WORK

We now discuss related work in the areas of MapReduce, iterative optimization, and measuring interference in multicore systems.

MapReduce. MapReduce [Dean and Ghemawat 2004] has been applied to a broad set of algorithms since its introduction [Das et al. 2007; Liu et al. 2010; Ranger et al. 2007; Verma et al. 2009]. Aside from Google's proprietary implementation of MapReduce, other open-source implementations include Hadoop [Hadoop 2014] and Sector/Sphere [Gu and Grossman 2009]. 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.

Iterative optimization. There exists a large body of research on iterative optimization [Agakov et al. 2006; Cavazos et al. 2007; Franke et al. 2005; Pan and Eigenmann 2006; Fursin et al. 2007; Cooper et al. 1999, 2005; Stephenson and Amarasinghe 2005]. Most of the prior work in iterative optimization assumes a single or a limited number of datasets. A few studies investigate input dataset sensitivity [Berube and Amaral 2006; Haneda et al. 2006; Fursin et al. 2007]. Recently, Chen et al. [2010] proposed KDataSets, a suite of 1,000 datasets per program, for evaluating iterative optimization with respect to dataset sensitivity. They conclude 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 dataset. The program-optimal combination was chosen after the fact though, with the assumption that all datasets are known, which we could not rely upon in the online system proposed in this article. Many studies [Kulkarni et al. 2004; Qasem et al. 2006; Agakov et al. 2006] focus on how to search the optimization space quickly and effectively using machine learning techniques. These studies search the optimization space offline without considering the practical issues for the online scenario on which we focus (e.g., compilation cost, dataset sensitivity, performance noise). A number of studies employ iterative optimization to optimize performance during runtime. Voss and Eigenmann [2000] propose ADAPT, which compiles code sections into different versions and which

dynamically selects the best-performing ones during runtime to adapt to different architectures and inputs. Fursin and Temam [2009] compile the most time-consuming routines into multiple versions using different combinations, and they use statistics to choose the best-performing version. Because of the limited number of input datasets available, they have to reuse data sets. Stephenson [2006] proposes a similar approach within a Java virtual machine. In contrast to this collection of prior work, IODC carefully manages costs versus benefits, enabling online iterative optimization in the data center while being performance noise aware. In addition, IODC explores a larger combination space of compiler optimizations during runtime.

Measuring interference. A flurry of recent work has considered performance interference through shared resources in multicore processors (e.g., Mars et al. [2010, 2011], Kambadur et al. [2012], Lim et al. [2012], and Dwyer et al. [2012]). Mars et al. [2011] present Bubble-Up, a characterization methodology to predict performance degradations from co-runners by applying a tunable amount of "pressure" to the memory subsystem in a data center context. Kambadur et al. [2012] introduce measurement techniques for analyzing interference in live production data center environments. In this work, we measure the impact of interference through the iterative optimization runtime by comparing sampled runs in isolation versus runs with co-runners. To the best of our knowledge, there is no prior work that has looked into how performance noise caused by co-runners affects the effectiveness of iterative optimization.

9. CONCLUSION AND FUTURE WORK

Implementing iterative optimization in an efficient way so that it can be deployed in an online environment is nontrivial for a number of practical hurdles. In this article, we propose IODC and show that the data center offers a context in which all practical hurdles can be overcome. 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 and compute-intensive throughput server applications, and it is completely transparent to the end user. We report an average performance improvement of 1.48×, and up to 2.08×, for five typical MapReduce applications, and 1.12×, and up to 1.39×, for nine typical server applications. As a second step, we evaluate IODC in the presence of performance noise due to co-runners. The results show that IODC is robust enough to find well-performing combinations of compiler optimizations in most cases. We enhance IODC with the ability of finding compatible co-schedules along with a threshold-based mechanism to dynamically control IODC's aggressiveness, which can achieve an up to 13% performance improvement over the worst possible co-runner's schedule.

As part of our future work, we plan to integrate more techniques (e.g., machine learning, check pointing) to further accelerate the search process of the huge compiler optimization space. We also intend to improve the IODC strategy in the following two aspects: (1) we will factor in the confidence level (used in the speedup comparison) as a parameter to help overcome performance noise and evaluate IODC using more co-running application pairs; (2) we would like to implement some scheduling heuristics with DMI to gradually learn better schemes in a co-running scenario. Finally, we will study other possible practical issues to further improve the effectiveness and robustness of IODC in a real data center context.

APPENDIX

A. 127 compiler options

Of the 127 GCC 4.4 compiler options mentioned in Section 4.3, 13 control the parameters of some optimization passes. They are listed in Table V, with the value range of the

Table V. Compiler Options (13 in Total) for Which We Need to Specify a Value from a Specific Range

Flag	Range
–param max-unrolled-insns=	32–2,048
–param max-unroll-times=	2–16
–param ira-max-loops-num=	32–256
–param large-stack-frame-growth=	512–2,048
–param max-inline-insns-recursive=	128–1,024
–param max-inline-insns-recursive-auto=	128–1,024
-fsched-stalled-insns-dep=	0–64
-fsched-stalled-insns=	0–64
-falign-functions=	1–64
-falign-jumps=	1–64
-falign-labels=	1–64
-falign-loops=	1–64
-ftree-parallelize-loops=	1–64

parameters explored. Table VI lists the rest of the options that turn on/off optimization passes. To create a combination of compiler options, we always use `-O3/-O2/-Os` as the first option, then randomly choose other options from all of these 127 options; the total number of options selected for a particular combination is also determined at random.

Table VI. Compiler Options (114 in Total) that can be Turned on/off

(-f/-fno-)align-functions	(-f/-fno-)align-jumps
(-f/-fno-)align-labels	(-f/-fno-)align-loops
(-f/-fno-)branch-count-reg	(-f/-fno-)branch-target-load-optimize
(-f/-fno-)branch-target-load-optimize2	(-f/-fno-)btr-bb-exclusive
(-f/-fno-)caller-saves	(-f/-fno-)conserve-stack
(-f/-fno-)cprop-registers	(-f/-fno-)crossjumping
(-f/-fno-)cse-follow-jumps	(-f/-fno-)cse-skip-blocks
(-f/-fno-)dce	(-f/-fno-)defer-pop
(-f/-fno-)delayed-branch	(-f/-fno-)delete-null-pointer-checks
(-f/-fno-)dse	(-f/-fno-)early-inlining
(-f/-fno-)expensive-optimizations	(-f/-fno-)forward-propagate
(-f/-fno-)function-cse	(-f/-fno-)gcse
(-f/-fno-)gcse-after-reload	(-f/-fno-)gcse-las
(-f/-fno-)gcse-lm	(-f/-fno-)gcse-sm
(-f/-fno-)guess-branch-probability	(-f/-fno-)if-conversion
(-f/-fno-)if-conversion2	(-f/-fno-)indirect-inlining
(-f/-fno-)inline-functions	(-f/-fno-)inline-functions-called-once
(-f/-fno-)inline-small-functions	(-f/-fno-)ipa-cp
(-f/-fno-)ipa-cp-clone	(-f/-fno-)ipa-matrix-reorg
(-f/-fno-)ipa-pta	(-f/-fno-)ipa-pure-const
(-f/-fno-)ipa-reference	(-f/-fno-)ipa-struct-reorg
(-f/-fno-)ira-coalesce	(-f/-fno-)ira-share-save-slots
(-f/-fno-)ira-share-spill-slots	(-f/-fno-)ivopts
(-f/-fno-)loop-block	(-f/-fno-)loop-interchange
(-f/-fno-)loop-strip-mine	(-f/-fno-)merge-constants
(-f/-fno-)modulo-sched	(-f/-fno-)modulo-sched-allow-regmoves
(-f/-fno-)move-loop-invariants	(-f/-fno-)omit-frame-pointer
(-f/-fno-)optimize-sibling-calls	(-f/-fno-)peel-loops
(-f/-fno-)peephole	(-f/-fno-)peephole2
(-f/-fno-)predictive-commoning	(-f/-fno-)prefetch-loop-arrays
(-f/-fno-)regmove	(-f/-fno-)rename-registers
(-f/-fno-)reorder-blocks	(-f/-fno-)reorder-blocks-and-partition
(-f/-fno-)reorder-functions	(-f/-fno-)rerun-cse-after-loop

(Continued)

Table VI. Continued

(-f/-fno-)reschedule-modulo-scheduled-loops	(-f/-fno-)rtl-abstract-sequences
(-f/-fno-)sched-interblock	(-f/-fno-)sched-spec
(-f/-fno-)sched-spec-load	(-f/-fno-)sched-spec-load-dangerous
(-f/-fno-)sched2-use-superblocks	(-f/-fno-)schedule-insns
(-f/-fno-)schedule-insns2	(-f/-fno-)see
(-f/-fno-)sel-sched-pipelining	(-f/-fno-)sel-sched-pipelining-outer-loops
(-f/-fno-)selective-scheduling	(-f/-fno-)selective-scheduling2
(-f/-fno-)split-ivs-in-unroller	(-f/-fno-)split-wide-types
(-f/-fno-)strict-aliasing	(-f/-fno-)strict-overflow
(-f/-fno-)thread-jumps	(-f/-fno-)tracer
(-f/-fno-)tree-builtin-call-dce	(-f/-fno-)tree-ccp
(-f/-fno-)tree-ch	(-f/-fno-)tree-copy-prop
(-f/-fno-)tree-copyrename	(-f/-fno-)tree-dce
(-f/-fno-)tree-dominator-opts	(-f/-fno-)tree-dse
(-f/-fno-)tree-fre	(-f/-fno-)tree-loop-distribution
(-f/-fno-)tree-loop-im	(-f/-fno-)tree-loop-ivcanon
(-f/-fno-)tree-loop-linear	(-f/-fno-)tree-loop-optimize
(-f/-fno-)tree-pre	(-f/-fno-)tree-reassoc
(-f/-fno-)tree-sink	(-f/-fno-)tree-sra
(-f/-fno-)tree-switch-conversion	(-f/-fno-)tree-ter
(-f/-fno-)tree-vect-loop-version	(-f/-fno-)tree-vectorize
(-f/-fno-)tree-vrp	(-f/-fno-)unroll-all-loops
(-f/-fno-)unswitch-loops	(-f/-fno-)variable-expansion-in-unroller
(-f/-fno-)vect-cost-model	(-f/-fno-)web

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. 295–305.

N. Aghdaie and Y. Tamir. 2002. 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'02)*. IEEE, Los Alamitos, CA, 63–68.

R. Agrawal and R. Srikant. 1994. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94)*. 487–499.

P. Berube and J. N. Amaral. 2006. Aestimo: A feedback-directed optimization evaluation tool. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'06)*. 251–260.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81.

J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. 2007. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*. 185–197.

Y. Chen, S. Fang, L. Eeckhout, O. Temam, and C. Wu. 2012. Iterative optimization for the data center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 49–60.

Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. 2010. Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*. 448–459.

K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. 2005. ACME: Adaptive compilation made efficient. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*. 69–77.

K. D. Cooper, P. J. Schielke, and D. Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*. 1–9.

A. S. Das, M. Datar, A. Garg, and S. Rajaram. 2007. Google news personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. 271–280.

J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. 107–113.

T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei. 2012. A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. IEEE, Los Alamitos, CA, 1–11.

B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. 2005. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*. 78–86.

G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. 2007. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'07)*. 245–260.

G. Fursin and O. Temam. 2009. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09)*. 34–49.

D. Gillick, A. Faria, and J. DeNero. 2006. *MapReduce: Distributed Computing for Machine Learning*. Technical Report 1–12. University of California, Berkeley, CA.

Y. Gu and R. L. Grossman. 2009. Sector and sphere: The design and implementation of a high performance data cloud. *Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure*. 367, 1897, 2429–2445.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual International Workshop on Workload Characterization (WWC'01)*. 3–14.

Hadoop. 2014. Apache Hadoop Home Page. Retrieved April 6, 2015, from http://hadoop.apache.org.

M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. 2006. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. 385–385.

K. Hoste and L. Eeckhout. 2008. Cole: Compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. 165–174.

M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. 2012. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. 1–12.

P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. 2004. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. 171–182.

S. Lim, J. Huh, Y. Kim, G. M. Shipman, and C. R. Das. 2012. D-factor: A quantitative model of application slow-down in multi-resource shared systems. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*. 271–282.

Z. Liu, H. Li, and G. Miao. 2010. MapReduce-based backpropagation neural network over large scale mobile data. In *Proceedings of the 6th International Conference on Natural Computation (ICNC'10)*. 1726–1730.

Loongson. 2014. Loongson 2F. Retrieved April 6, 2015, from http://www.loongson.cn/.

J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. 2011. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the International Symposium on Microarchitecture (MICRO'11)*. 248–259.

J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. 2010. Contention aware execution: Online contention detection and response. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'10)*. 257–265.

M. Marwah, S. Mishra, and C. Fetzer. 2008. 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'08)*. 167–176.

MovieLens. 2014. MovieLens Data Sets. Retrieved April 6, 2015, from http://www.grouplens.org/node/73.

M. Nawaz, E. E. Enscore Jr., and I. Ham. 1983. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *OMEGA* 11, 1, 91–95.

Z. Pan and R. Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. 319–332.

PAPI 5.1. 2013. PAPI: Performance Application Programming Interface. Retrieved April 6, 2015, from http://icl.cs.utk.edu/papi.

S. Patil and D. J. Lilja. 2010. Using resampling techniques to compute confidence intervals for the harmonic mean of rate-based performance metrics. *Computer Architecture Letters* 9, 1, 1–4.

A. Qasem, K. Kennedy, and J. Mellor-Crummey. 2006. Automatic tuning of whole applications using direct search and a performance-based transformation system. *Journal of Supercomputing* 36, 2, 183–196.

C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. 13–24.

M. Stephenson. 2006. *Automating the Construction of Compiler Heuristics Using Machine Learning*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. 123–134.

M. Stephenson, M. Martin, and U. M. O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. 77–90.

L. Tang, J. Mars, and M. L. Soffa. 2011. Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*. 12–21.

A. Verma, X. Llora, D. E. Goldberg, and R. H. Campbell. 2009. Scaling genetic algorithms using MapReduce. In *Proceedings of the 9th International Conference on Intelligent Systems Design and Applications (ISDA'09)*. 13–18.

M. J. Voss and R. Eigenmann. 2000. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP'00)*. 163–170.

Received October 2014; revised January 2015; accepted February 2015