A Practical Method For Quickly Evaluating Program Optimizations

Grigori Fursin,[1,2] Albert Cohen,[1] Michael O'Boyle[2] and Olivier Temam[1]

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

Abstract. This article aims at making iterative optimization practical and usable by speeding up the evaluation of a large range of optimizations. Instead of using a full run to evaluate a single program optimization, we take advantage of periods of stable performance, called phases. For that purpose, we propose a low-overhead phase detection scheme geared toward fast optimization space pruning, using code instrumentation and versioning implemented in a production compiler.

Our approach is driven by simplicity and practicality. We show that a simple phase detection scheme can be sufficient for optimization space pruning. We also show it is possible to search for complex optimizations at run-time without resorting to sophisticated dynamic compilation frameworks. Beyond iterative optimization, our approach also enables one to quickly design self-tuned applications.

Considering 5 representative SpecFP2000 benchmarks, our approach speeds up iterative search for the best program optimizations by a factor of 32 to 962. Phase prediction is 99.4% accurate on average, with an overhead of only 2.6%. The resulting self-tuned implementations bring an average speed-up of 1.4.

1 Introduction

Recently, iterative optimization has become an increasingly popular approach for tackling the growing complexity of processor architectures. Bodin et al. [7] and Kisuki et al. [22] have initially demonstrated that exhaustively searching an optimization parameter space can bring performance improvements higher than the best existing static models, Cooper et al. [13] have provided additional evidence for finding best sequences of various compiler transformations. Since then, recent studies [34, 19] demonstrate the potential of iterative optimization for a large range of optimization techniques.

Some studies show how iterative optimization can be used *in practice*, for instance, for tuning optimization parameters in libraries [38, 6] or for building static models for compiler optimization parameters. Such models derive from the automatic discovery of the mapping function between key program characteristics and compiler optimization parameters; e.g., Stephenson et al. [32] successfully applied this approach to unrolling.

However, most other articles on iterative optimization take the same approach: several benchmarks are repeatedly executed with the same data set, a new optimization parameter (e.g., tile size, unrolling factor, inlining decision,. . .) being tested at each execution. So, while these studies demonstrate the *potential* for iterative optimization, few provide a *practical* approach for effectively applying iterative optimization. The issue at stake is: what do we need to do to make iterative optimization a reality? There are three main caveats to iterative optimization: quickly scanning a large search space, optimizing based on and across multiple data sets, and extending iterative optimization to complex composed optimizations beyond simple optimization parameter tuning.

In this article, we aim at the general goal of making iterative optimization a usable technique and especially focus on the first issue, i.e., how to speed up the scanning of a large optimization space. As iterative optimization moves beyond simple parameter tuning to composition of multiple transformations [19, 26, 11] (the third issue mentioned above), this search space can become potentially huge, calling for faster evaluation techniques. There are two possible ways to speeding up the search space scanning: search more smartly by exploring points with the highest potential using genetic algorithms and machine learning techniques [12, 13, 35, 33, 3, 25, 20, 32], or scan more points within the same amount of time. Up to now, speeding up the search has mostly focused on the former approach, while this article is focused on the latter one.

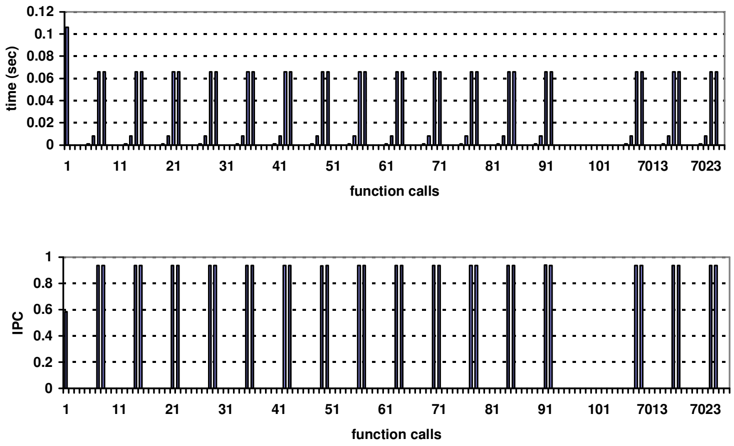

Fig. 1. *Execution time and IPC for subroutine* `resid` *of benchmark* `mgrid` *across calls.*

The principle of our approach is to improve the efficiency of iterative optimization by taking advantage of program *performance stability* at run-time. There is ample evidence that many programs exhibit phases [30, 23], i.e., program trace intervals of several millions instructions where performance is similar. What is the point of waiting for the end of the execution in order to evaluate an optimization decision (e.g., evaluating a tiling or unrolling factor, or a given composition of transformations) if the program performance is stable within phases or the whole execution? One could take advantage of phase intervals with the same performance to evaluate a different optimization option at each interval. As in standard iterative optimization, many options are evaluated, except that multiple options are evaluated within the same run.

The main assets of our approach over previous techniques are simplicity and practicality. We show that, for many benchmarks, a low-overhead performance stability/phase detection scheme is sufficient for optimization space pruning. We also show that it is possible to search (even complex) optimizations at run-time without resorting to sophisticated dynamic optimization/recompilation frameworks. Beyond iterative optimization, our approach also enables one to quickly design self-tuned applications, significantly easier than current manually tuned libraries.

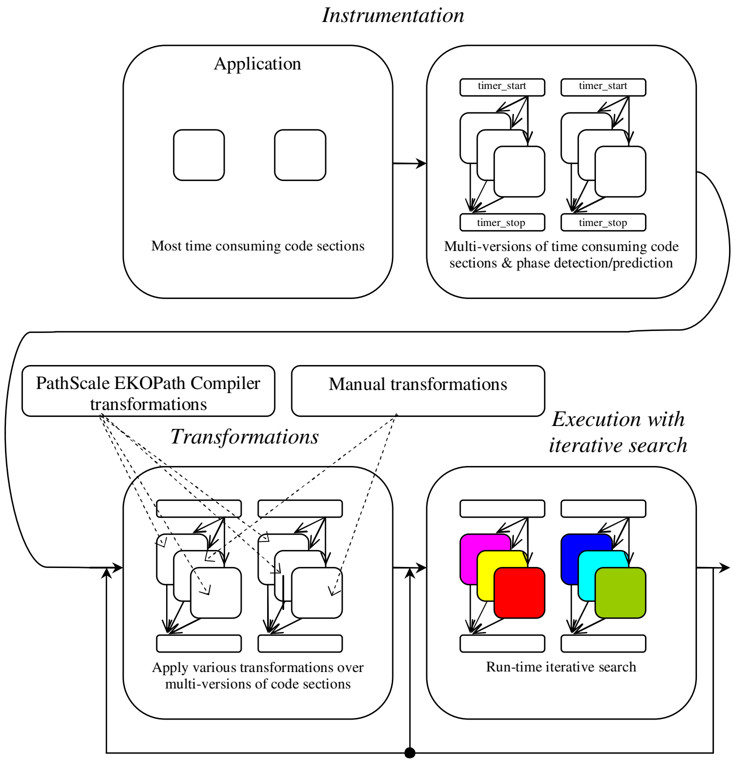

Fig. 2. *Application instrumentation and multi-versioning for run-time iterative optimization*

Phase detection and optimization evaluation are respectively implemented using code instrumentation and versioning within the EKOPath compiler. Considering 5 self-tuned SpecFP2000 benchmarks, our space pruning approach speeds up iterative search by a factor of 32 to 962, with a 99.4% accurate phase prediction and a 2.6% performance overhead on average; we achieve speedups ranging from 1.10 to 1.72.

The paper is structured as follows. Section 2 provides the motivation, showing how our technique can speedup iterative optimization, and including a brief description of how it may be applied in different contexts. Section 3 describes our novel approach to runtime program stability detection. This is followed in Section 4 by a description of our dynamic transformation evaluation technique. Section 5 describes the results of applying these techniques to well known benchmarks and is followed in Section 6 by a brief survey of related work. Section 7 concludes the paper.

2 Motivation

This section provide a motivating example for our technique and outlines the ways in which it can be used in program optimization.

2.1 Example

Let us consider the `mgrid` SpecFP2000 benchmark. For the sake of simplicity, we have tested only 16 random combinations of traditional transformations, known to be efficient, on the two most time consuming subroutines `resid` and `psinv`. These transformations include loop fusion/fission, loop interchange, loop and register tiling, loop unrolling, prefetching. Since the original execution time of `mgrid` is 290 seconds (for the reference data set), a typical iterative approach for selecting the best optimization option would take approximately $290 \times 32 = 9280$ seconds (more than 2 hours). Moreover, all these tests are conducted with the same data set, which does not make much sense from a practical point of view.

However, considering the execution time of every call to the original subroutine `resid` in Figure 1, one notices fairly stable performance across pairs of consecutive calls with period 7.[1] Therefore, we propose to conduct most of these iterations at runtime, evaluating multiple versions during a single or a few runs of the application. The overall iterative program optimization scheme is depicted in Figure 2.

Practically, we insert all 16 different optimized versions of `resid` and `psinv` into the original code. As shown in the second box of Figure 2, each version is enclosed by calls to monitoring functions before and after the instrumented section. These timer functions monitor the execution time and performance of any active subroutine version using the high-precision PAPI hardware counters library [8], allowing to switch at runtime among the different versions of this subroutine. This low-overhead instrumentation barely skews the program execution time (less than 1%) as shown in Figure 3.

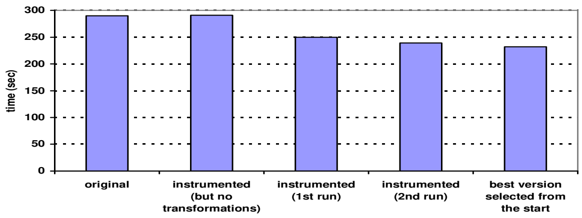

Fig. 3. *Execution times for different versions of benchmark* `mgrid`

If one run is not enough to optimize the application, it is possible to iterate on the multi-version program, the fourth box in Figure 2. Eventually, if new program transformations need to be evaluated, or when releasing an optimized application restricted to the most effective optimizations, one may also iterate back to apply a new set of transformations, the third box in Figure 2.

Figure 4 details the instrumentation and versioning scheme. Besides starting and stopping performance monitoring, `timer_start` and `timer_stop` have two more functions: `timer_stop` detects performance stability for consecutive or periodic executions of the selected section, using execution time and IPC; then `timer_start` predicts that performance will remain stable, in order to evaluate and compare new options. After stability is detected, `timer_start` redirects execution sequentially to the optimized versions of the original subroutine. When the currently evaluated version has exhibited stable performance for a few executions (2 in our case), we can measure its impact on performance *if* the phase did not change in the meantime. To validate this, the original

[1] Calls that take less than 0.01s are ignored to avoid startup or instrumentation overhead, therefore their IPC bars are not shown in this figure.

Original code

```
      SUBROUTINE RESID(U,V,R,N,A)
      REAL*8 U(N,N,N),V(N,N,N),R(N,N,N),A(0:3)
      INTEGER N, I3, I2, I1
        BODY OF THE SUBROUTINE
      RETURN
      END
```

Instrumented code

```
      SUBROUTINE RESID(U,V,R,N,A)
      REAL*8 U(N,N,N),V(N,N,N),R(N,N,N),A(0:3)
      INTEGER N, I3, I2, I1
      INTEGER FSELECT

      CALL TIMER_START(00001, FSELECT)
      GOTO (1100, 1101, 1102, 1103, 1104, 1105,
        (...)
     +1115, 1116), FSELECT+1
        (...)
1100  CONTINUE
      CALL RESID_00(U,V,R,N,A)
      GOTO 1199
1101  CONTINUE
      CALL RESID_01(U,V,R,N,A)
      GOTO 1199
        (...)
1199  CONTINUE
      CALL TIMER_STOP(00001)
      RETURN
      END

      SUBROUTINE RESID_00(U,V,R,N,A)
      REAL*8 U(N,N,N),V(N,N,N),R(N,N,N),A(0:3)
      INTEGER N, I3, I2, I1
        BODY OF THE SUBROUTINE
      RETURN
      END

      SUBROUTINE RESID_01(U,V,R,N,A)
      REAL*8 U(N,N,N),V(N,N,N),R(N,N,N),A(0:3)
      INTEGER N, I3, I2, I1
        BODY OF THE SUBROUTINE
      RETURN
      END
```

Fig. 4. *Instrumentation example for subroutine* `resid` *of benchmark* `mgrid`*.*

code is executed again a few times (2 in our case to avoid transitional effects). In the same way all 16 versions are evaluated during program execution and the best one is selected at the end, as shown in Figure 5a.

Overall, evaluating all 16 optimization options for subroutine `resid` requires only 17 seconds instead of 9280 thus speeding up iterative search 546 times. Furthermore, since the best optimization has been found after only 6% of the code has been executed, the remainder of the execution uses the best optimization option and the overall `mgrid` execution time is improved by 13.7% all in one run (one data set) as shown in Figure 5b. The results containing original execution time, IPC and the corresponding best option which included loop blocking, unrolling and prefetching in our example, is saved in the database after the program execution. Therefore, during a second run with the same dataset (assuming standard across-runs iterative optimization), the best optimization option is selected immediately after the period is detected and the overall execution time is improved by 16.1% as shown in Figure 5c. If a different dataset is used and the behavior of the program changed, the new best option will be found for this context and saved into the database. Finally, the execution time of the non-instrumented code with the best version implemented from the start (no run-time convergence) brings almost the same performance of 17.2% as shown in Figure 5d and 3. The spikes on the graphs in Figure 5b,c are due to the periodic change in calling context of subroutine `resid`. At such change points, the phase detection mechanism produces a miss and starts executing *the original non-transformed version* of the subroutine, to quickly detect the continuation of this phase or the beginning of another one (new or with a known behavior).

2.2 Application scenarios

The previous example illustrates the two main applications of our approach. The first one is iterative optimization, and the second is dynamic self-tuning code.

In the first case, each run — and each phase within this run — exercises multiple optimization options, including complex sequences of compiler or manual transformations. The phase analysis and optimization decisions are dumped into a database. This facility can be used for across-runs iterative optimization. There are two cases where this approach is practical. First, some applications may exhibit similar performance across multiple data sets, providing key parameters do not change (e.g., matrix dimen-

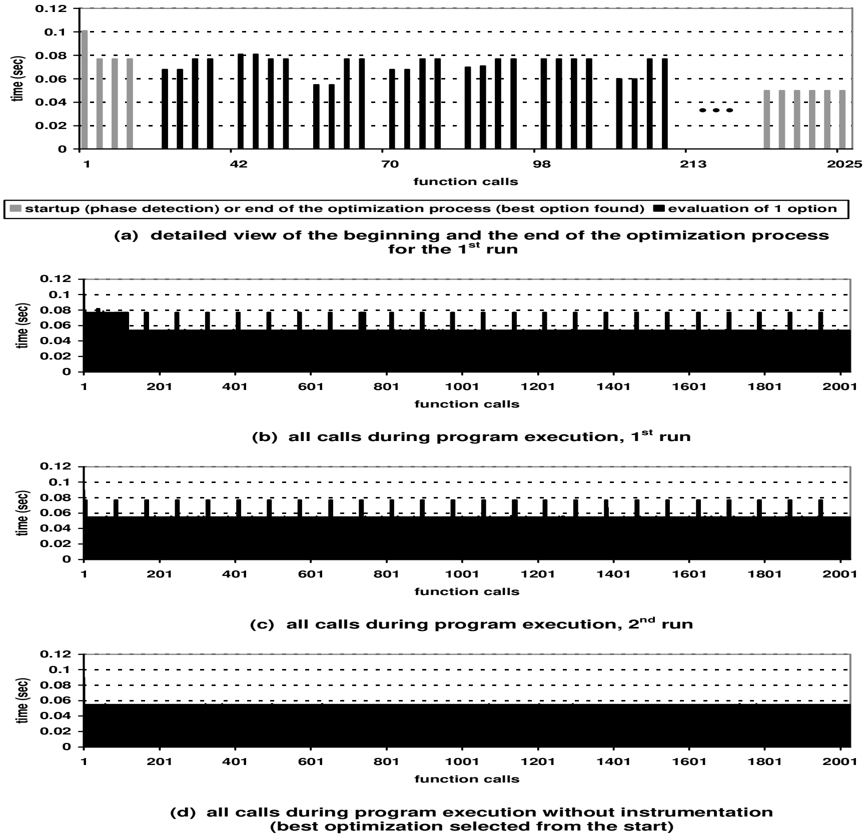

Fig. 5. *Execution times for subroutine* `resid` *of benchmark* `mgrid` *during run-time optimization*

sions do not change, but matrix values do); second, even when the performance of a code section varies with the data set, it is likely that a few optimizations will be able to achieve good results for a large range of data sets. In both cases, run-time iterative optimization can speed up optimization space search: a selection of optimizations is evaluated during each run, progressively converging to the best optimization.

In the second case, our technique allows to create self-tuning programs which adjust to the current data set, within a production run. Assuming the optimized versions known to perform best (in general, for multiple benchmarks) have been progressively learned across previous runs and data sets, one can implement a self-tuning code by selecting only a few of those versions. Even if some of the selected versions perform poorly, they do not really affect overall execution time since convergence occurs quickly and most of the execution time is spent within the best ones.

3 Dynamic Stability Prediction

The two key difficulties with dynamic iterative optimization are how to evaluate multiple optimization options at run-time and when can they be evaluated. This section tackles the second problem by detecting and predicting stable regions of the program where optimizations may be evaluated.

3.1 Performance stability and phases

As mentioned in the introduction, multiple studies [30, 27] have highlighted that programs exhibit *phases*, i.e., performance can remain stable for many millions instructions and performance patterns can recur within the program execution. Phase analysis is now extensively used for selecting sampling intervals in processor architecture simulation, such as in SimPoint [27]. More recently, phase-based analysis has been used to tune program power optimizations by dynamically adapting sizes of L1 and L2 caches [21].

For iterative optimization, phases mean that the performance of a given code section will remain stable for multiple consecutive or periodic executions of that code section. One can take advantage of this stability to compare the effect of multiple different optimization options. For instance, assuming one knows that two consecutive executions $E1$ and $E2$ of a code section will exhibit the same performance P, one can collect P in $E1$, apply a program transformation to the code section, and collect its performance P' in $E2$; by comparing P and P', one can decide if the program transformation is useful. Obviously, this comparison makes sense only if $E1$ and $E2$ exhibit the same baseline performance P, i.e., if $E1$ and $E2$ belong to the same phase. So, the key is to detect when phases occur, i.e., where are the regions with identical baseline performance. Also, IPC may not always be a sufficient performance metric, because some program transformations may increase or reduce the number of instructions, such as unrolling or scalar promotion. Therefore, we monitor not only performance stability but also execution time stability across calls, depending on the program transformations.

Figures 6 and 1 illustrate IPC and execution time stability of one representative subroutine for 5 SpecFP2000 benchmarks by showing variations across calls to the same subroutine. These benchmarks are selected to demonstrate various representative behavior for floating point programs. For instance, the `applu` subroutine has a stable performance across all calls except for the first one; the `galgel` subroutine has periodic performance changes with 5 shifts during overall execution; the `equake` most time-consuming section exhibits unstable performance for 250 calls and then becomes stable; the `apsi` subroutine has a stable performance across all calls; finally, the `mgrid` subroutine exhibits periodic stable performance.

Detecting stability For the moment, we do not consider program transformations, and two instances are compared solely using IPC. We then simply define stability by 3 consecutive of periodic code section execution instances with the same IPC. Naturally, this stability characterization is speculative, the 4th instance performance may vary, but in practice as graphs in Figure 6 suggest, stability regions are long and regular enough so that the probability of incorrect stability detections (miss rate) is fairly low.

Note however, that the occurrence of a phase (an execution instance with a given performance) is only detected *after* it has occurred, i.e., when the counter value is collected and the code section instance already executed. If changes in the calling context occur faster than the evaluation of a new optimization option, there may not be long enough consecutive executions with stable performance to measure the impact of the optimization. Therefore, it is not sufficient to *react* to phase changes: within a phase, it

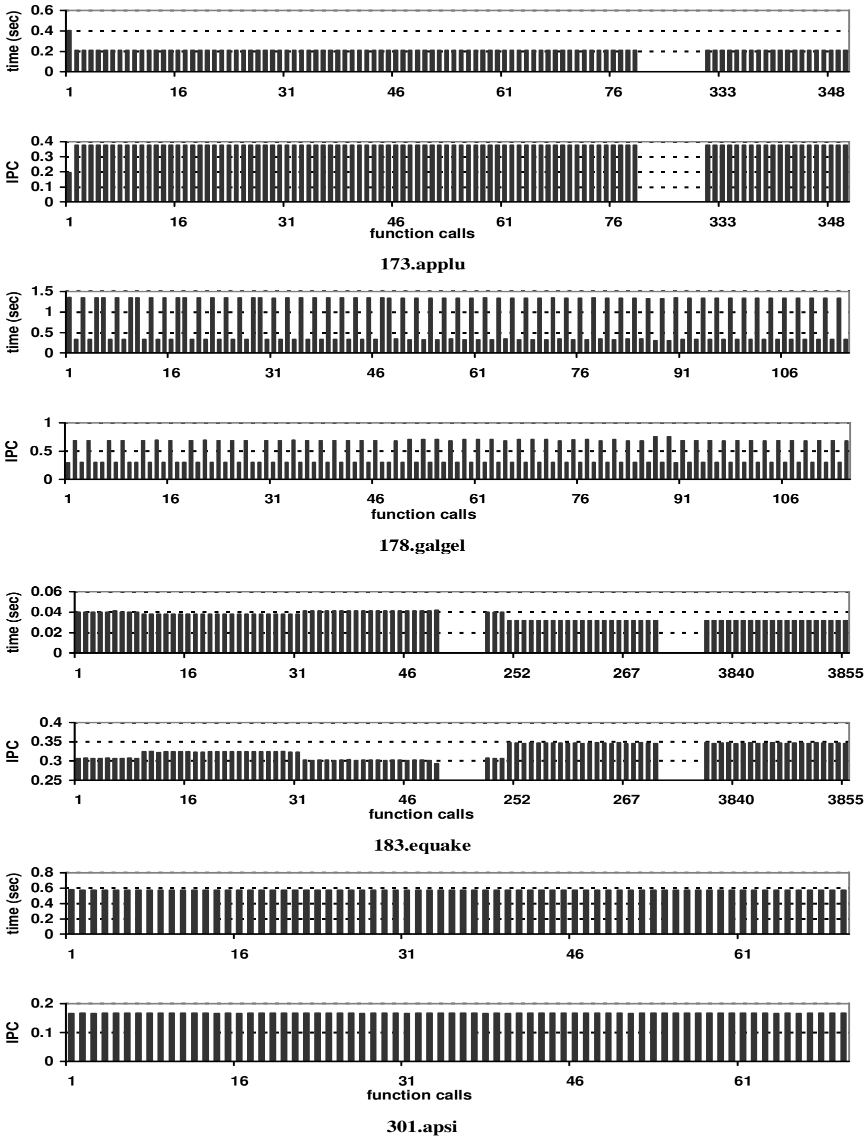

Fig. 6. *Execution time and IPC for one representative subroutine per benchmark (across calls)*

is necessary to detect the *length of consecutive regions* of stable performance and to *predict* their occurrence. Fortunately, Figure 6 shows that phases tend to recur regularly, especially in scientific applications which usually have simple control flow behavior, which is further confirmed by other broader experiments [31].

To predict the occurrence of regular phases, for each instrumented code section, we store the performance measurement along with the number of calls exhibiting the same performance (phase length) in a Phase Detection and Prediction Table (PDPT) as

Application	Code sections	Phases	Hits	Misses	Miss rate
mgrid	a	1	1924	27	0.014
	b	1	998	1	0.001
applu	a	1	348	0	0
	b	2	349	0	0
	c	2	349	0	0
	d	1	350	0	0
	e	1	350	0	0
galgel	a	2	86	12	0.140
	b	2	83	14	0.169
equake	a	2	3853	1	0.000
apsi	a	1	69	0	0
	b	1	69	0	0
	c	1	69	0	0
	d	1	69	0	0
	e	1	70	0	0
	f	1	69	0	0

Table 1. *Number of phases, hits and misses per code section for each application.*

shown in Figure 7. If a performance variation occured, we check the table to see if a phase with such behavior already occurred, and if so, we also record the distance (in number of calls) since it occurred. At the third occurrence of the same performance behavior, we conclude the phase becomes stable and recurs regularly, i.e., with a fixed period, and we can predict its next occurrence. Then, program transformations are applied to the code section, and the performance effects are only compared within the same phase, i.e., for the same baseline performance, avoiding to reset the search each time the phase changes. The length parameter indicates when the phase will change. Thus, the transformation space is searched independently for all phases of a code section. This property has the added benefit of allowing per-phase optimization, i.e., converging towards different optimizations for different phases of the same code section.

Table 1 shows how the phase prediction scheme performs. Due to the high regularity of scientific applications, our simple phase prediction scheme has a miss rate lower than 1.4% in most of the cases, except for `galgel` which exhibits miss rates of 14% and 17% for two time-consuming subroutines. Also, note that we assumed two performance measurements were identical provided they differ by less than a threshold determined by observed measurement error, of the order of 2% with our experimental environment.

3.2 Compiler instrumentation

Since, program transformations target specific code sections, phase detection should target code sections rather than the whole program. In order to monitor code sections performance, we instrument a code section, e.g., a loop nest or a function, with performance counter calls from the hardware counters PAPI library. Figure 4 shows an example instrumentation at the subroutine/function level for `mgrid` (Fortran 77) and its `resid` subroutine. Figure 7 shows the details of our instrumentation.

Each instrumented code section gets a unique identifier, and before and after each section, monitoring routines `timer_start` and `timer_stop` are called. These routines record the number of cycles and number of instructions to compute the IPC (the first argument is the unique identifier of the section). At the same time, `timer_stop` detects phases and stability, and `timer_start` decides which optimization option should be evaluated next and returns variable `FSELECT` to branch to the appropriate optimization option (versioning), see the `GOTO` statement.

Instrumentation is currently applied before and after the call functions and the outer loops of all loop nests with depth 2 or more (though the approach is naturally useful for the most time-consuming loop nests and functions only). Note that instrumented loop nests can themselves contain subroutine calls to evaluate inlining; however we forbid nested instrumentations, so we systematically remove outer instrumentations if nested calls correspond to loop nests.

4 Evaluating Optimizations

Once a stable period has been detected we need a mechanism to evaluate program transformations and evaluate their worth.

4.1 Comparing optimization options

As soon as performance stability is observed, the evaluation of optimization options starts. A new optimization option is said to be evaluated only after 2 consecutive executions with the same performance. The main issue is to combine the detection of phases with the evaluation of optimizations, because, if the phase detection scheme does not predict that a new phase starts, baseline performance will change, and we would not know whether performance variations are due to the optimization option being evaluated or to a new phase.

In order to verify the prediction, the instrumentation routine periodically checks whether baseline performance has changed (and in the process, it monitors the occurrence of new phases). After any optimization option evaluation, i.e., after 2 consecutive executions of the optimized version with the same performance, the code switches back to the original code section for two additional iterations. The first iteration is ignored to avoid transition effects because it can be argued that the previous optimized version of the code section can have performance side-effects that would skew the performance evaluation of the next iteration (the original code section). However, we did not find empirical evidence of such side-effects; most likely because code sections have to be long enough that instrumentation and start-up induces only a negligible overhead. If the performance of the second iteration is similar to the initial baseline performance, the effect of the current option is validated and further optimization options evaluation resumes. Therefore, evaluating an optimization option requires at least 4 executions of a given code section (2 for detecting optimization performance stability and 2 for checking baseline performance). For example, see the groups of black bars in Figure 5a of the motivation section (on benchmark `mgrid`). If the baseline performance is later found to differ, the optimization search for this other phase is restarted (or started if it is the first occurrence of that phase).[2]

Practically, the Phase Detection and Prediction Table (PDPT) shown in Figure 7 holds information about phases and their current state (detection and prediction), new

[2] Note that it is *restarted* not *reset*.

option evaluation or option validation (stability check). It also records the best option found for every phase.

4.2 Multiple evaluations at run-time

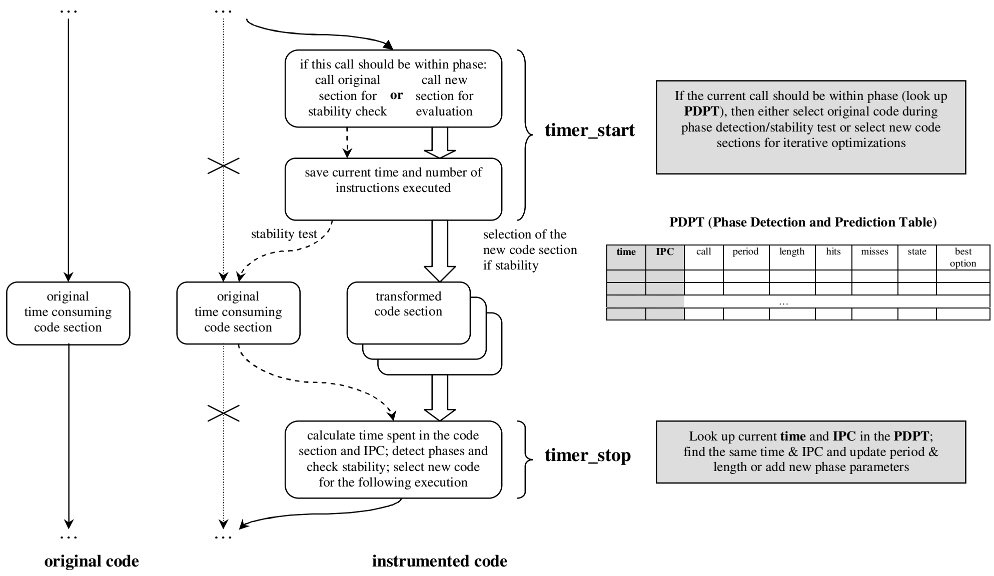

Fig. 7. *Code instrumentation for run-time adaptive iterative optimizations.*

Many optimizations are parameterized, e.g., tile size or unroll factor. However, in the context of run-time iterative optimization, whether changing a parameter just means changing a program variable (e.g., a parametric tile size), or changing the code structure (e.g., unroll factor) matters. The former type of optimization can be easily verified by updating the parameter variable. In order to accommodate the latter type of complex optimizations, we use versioning: we generate and optimize differently multiple versions of the same code section (usually a subroutine or a loop nest), plus the additional control/switching code driven by the monitoring routine as shown in Figures 7 and 4, using the EKOPath compiler.

The main drawback of versioning is obviously increased code size. While this issue matters for embedded applications, it should not be a serious inconvenience for desktop applications, provided the number of optimization options is not excessive. Considering only one subroutine or loop nest version will be active at any time, even the impact of versioning on instruction cache misses is limited. However, depending on what run-time adaptation is used for, the number of versions can vary greatly. If it is used for evaluating a large number of program transformations, including across runs, the greater the number of versions the better, and the only limitation is the code size increase. If it is used for creating self-adjusting codes that find the best option for the current run, it is best to limit the number of options, because if many options perform worse than the original version, the overall performance may either degrade or marginally improve. In our experiments, we limited the number of versions to 16.

This versioning scheme is simple but has practical benefits. Alongside the optimized versions generated by the compiler, the user can add subroutines or loop nests modified by hand, and either test them as is or combine them with compiler optimizations. User-suggested program transformations can often serve as starting points of the optimization search, and recent studies [15, 34] highlight the key role played by well selected starting points, adding to the benefit of combined optimizations. Moreover, another study [11] suggests that iterative optimization should not be restricted to program transformation parameter tuning, but should expand to selecting program transformations themselves, beyond the strict composition order imposed by the compiler. Versioning is a simple approach for testing a variety of program transformations compositions.

5 Experiments

The goal of this article is to speedup the evaluation of optimization options, rather than to speedup programs themselves. Still, we later report program speedups to highlight that the run-time overhead has no significant impact on program performance, that the run-time performance analysis strategy is capable of selecting appropriate and efficient optimization options, and that it can easily accommodate both traditional compiler-generated program transformations and user-defined ad-hoc program transformations.

5.1 Methodology

Platforms and tools. All experiments are conducted on an Intel Pentium 4 Northwood (ID9) Core at 2.4GHz (bus frequency of 533MHz), the L1 cache is 4-way 8KB, the L2 cache is 8-way 512KB, and 512MB of memory; the O/S is Linux SUSE 9.1. We use the latest PAPI hardware counter library [1] for program instrumentation and performance measurements. All programs are compiled with the open-source EKOPath 2.0 compiler and -Ofast flag [2], which, in average, performs similarly or better than the Intel 8.1 compiler for Linux.

Compiler-generated program transformations are applied using the EKOPath compiler. We have created an EKOPath API that triggers program transformations, using the compiler's optimization strategy as a starting point. Complementing the compiler strategy with iterative search enables to test a large set of combinations of transformations such as inlining, local padding, loop fusion/fission, loop interchange, loop/register tiling, loop unrolling and prefetching.

Target benchmarks. We considered five representative SpecFP2000 benchmarks with different behavior, as shown in Figure 6 (`mgrid`, `applu`, `galgel`, `equake`, `apsi`), using the `ref` data sets. We apply optimization only on the most-time consuming sections of these benchmarks. We handpicked these codes based on the study by Parello et al. [26] which suggests which SpecFP2000 benchmarks have the best potential for improvement (on an Alpha 21264 platform, though). Since the role of seed points in iterative search has been previously highlighted [3, 34], we also used the latter study as an indication for seed points, i.e., initial points for a space search.

5.2 Results

This section shows that the full power of iterative optimization can be achieved at the cost of profile-directed optimization: one or two runs of each benchmark are sufficient to discover the best optimization options for every phase.

Application	Max. number of potential evaluations	Number of options evaluated	Instrumentation overhead
mgrid	699	32	0%
applu	430	80	0.01%
galgel	32	32	0.01%
equake	962	16	13.17%
apsi	96	96	0%

Table 2. *Maximum number of potential evaluations or iterative search speedup vs. the real number of options evaluated during single execution and the associated overhead.*

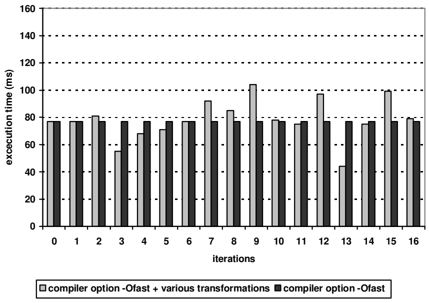

Fig. 8. *Execution time variations of the* `resid` *subroutine of the* `mgrid` *benchmark over iterations (subroutine calls).*

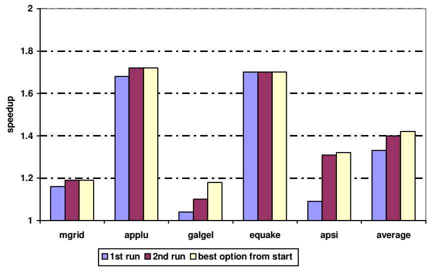

Fig. 9. *Speedups of instrumented program over original program after first run, second run or if the best option is selected from the start.*

Boosting search rate For each benchmark, Table 2 shows the actual number of evaluated options, which is the number of versions multiplied by the number of instrumented code sections and by the number of phases with significant execution time.

However, the maximum number of potential optimization options (program transformations or compositions of program transformations) that can be evaluated during one execution of a benchmark can be much higher depending on the application behavior. Thanks to run-time adaptation it is now possible to evaluate 32 to 962 more optimization options than through traditional across-runs iterative optimization. The discrepancy among maximum number of evaluations is explained by the differences in phase behavior of programs and in the instrumentation placement. If the instrumentation is located at a lower loop nest level, it enables a greater number of evaluations but it can also induce excessive overhead and may limit applicable transformations. On the other hand, instrumentation at a higher level enables all spectrum of complex sequences of transformations but can limit the number of potential evaluations during one execution. For example, the number of potential optimization evaluations is small for `galgel` due to chaotic behavior and frequent performance mispredictions, and is high for `equake` due to relatively low level instrumentation.

To quantify the instrumentation overhead, the last column in Table 2 shows the ratio of the execution time of the instrumented program over the original program, assuming all optimization options are replaced with exact copies of the original code section. As a result, this ratio only measures the slowdown due to instrumentation. The overhead is negligible for 4 out of 5 benchmarks, and reaches 13% for `equake`. Note however that `equake` still achieves one of the best speedups at 1.7, as shown in Figure 9.

Self-tuned speedup For each selected benchmark we have created a self-tuning program with 16 versions of each most time consuming sections of these benchmarks For each of these versions we applied either combinations of compiler-generated program transformations using our EKOPath compiler API (loop fusion/fission, loop interchange, loop and register tiling, loop unrolling and prefetching with multiple randomly selected parameters) or manual program transformations suggested by Parello et al. [26] for the Alpha platform, or combinations of both. The overall number of evaluations per each benchmark varied from 16 to 96 depending on the number of most-time consuming sections, as shown in table 2.

Figure 8 shows an example of execution time variations for the triple-nested loop in the `resid` subroutine of the `mgrid` benchmark for each option evaluation all within one execution of this program. The baseline performance is shown in a straight gray line and the best version is found at iteration 13. The final best sequence of transformations for the loop in the subroutine `resid` is loop tiling with tile size 60, loop unrolling with factor 16 and prefetching switched off. The best parameters found for subroutine `psinv` of the same benchmark are 9 for loop tiling and 14 for loop unrolling. Note also that the static algorithm of the EKOPath compiler suggested unrolling factors of 2 for loops of both subroutines. This factor is a kind of tradeoff value across all possible data sets, while the self-tuning code converged toward a different value dynamically. It is important to note that this adjustment occurred at run-time, during a single run, and that the optimization was selected soon enough to improve the remainder of the run.

Figure 9 shows the speedups obtained for all benchmarks after two executions as well as the speedups assuming there is no instrumentation overhead and the best optimization option is used from the start; speedups vary from 1.10 to 1.72. It is interesting to note that, though the manual transformations were designed for a different architecture (Alpha 21264), for 4 out of 5 benchmarks, they were eventually adjusted through transformation parameter tuning to our target architecture, and still perform well. In other terms, beyond performance improvement on a single platform, self-adjusting codes also provide a form of cross-platform portability by selecting the optimization option best suited for the new platform.

6 Related Work

Some of the first techniques to select differently optimized versions of code sections are cloning and multi-versioning [9, 14, 16]. They use simple version selection mechanisms according to the input run-time function or loop parameters. Such techniques are used to some extent in current compilers but lack flexibility and prediction and cannot cope with various cases where input parameters are too complex or differ while the behavior of the code section remains the same and vice versa.

To improve cloning effectiveness, many studies defer code versioning to the run-time execution. Program hot spots and input/context parameters are detected at run-time, to drive dynamic recompilation. For example, the Dynamo system [5] can optimize streams of native instructions at run-time using hot traces and can be easily implemented inside JIT compilers. Insertion of prefetching instructions or changing prefetching distance dynamically depending on hardware counters informations is presented in [28]. VCODE [18] is a tool to dynamically regenerate machine code, and [4] presents a technique which produces pre-optimized machine-code templates and later dynamically patch those templates with run-time constants. ADAPT [37, 36] applies high-level optimizations to program hot spots using dynamic recompilation in a separate process or on a separate workstation and describes a language to write self-tuned applications. Finally, ADORE [10, 24] uses a sampling based phase analysis to detect performance bottlenecks and apply simple transformations such as prefetching dynamically.

Recently, software-only solutions have been proposed to effectively detect, classify and predict phase transitions, with very low run-time overhead [5, 17]. In particular, [17] decouples this detection from the dynamic code generation or translation process, relying on a separate process sampling hardware performance counters at a fixed interval. Selection of a good sampling interval is critical, to avoid missing fine-grain phase changes while minimizing overhead [27, 29].

In contrast with the above-mentioned projects, our approach is a novel combination of versioning, code instrumentation and software-only phase detection to enable practical iterative evaluation of complex transformations at run-time. We choose static versioning rather than dynamic code generation, allowing low-overhead adaptability to program phases and input contexts. Associating static instrumentation and dynamic detection avoids most pitfalls of either isolated instrumentation-based or sampling-based phase analyses, including sensitivity to calling contexts and sampling interval selection [23]. Finally, we rely on predictive rather than reactive phase detection, although it is not for the reasons advocated in [17]: we do not have to amortize the overhead of run-time code generation, but we need to predict phase changes to improve the evaluation rate for new optimization options.

7 Conclusions and Perspectives

Several practical issues still prevent iterative optimization from being widely used, the time required to search the huge program transformations space being one of the main issues. In this article, we present a method for speeding up search space pruning by a factor of 32 to 962 over a set of benchmarks, by taking advantage of the phase behavior (performance stability) of applications. The method, implemented in the EKOPath compiler, can be readily applied to a large range of applications. The method has other benefits: such self-tuned programs facilitate portability across different architectures and software environments, they can self-adjust at the level of phases and to particular data sets (as opposed to the trade-off proposed by current iterative techniques), they can build a catalog of per-phase appropriate program transformations (code section/performance pairs) across runs, and they can easily combine user-suggested and compiler-suggested transformations thanks to their versioning approach.

Future work will include fast analysis of large complex transformation spaces, improving our phase detection and prediction scheme to capture more complex performance behaviors, and improving the instrumentation placement, especially using self-placement of instrumentation at the most proper loop nest levels, by instrumenting all loop nests levels, then dynamically switching off instrumentation at all levels but one, either using predication or versioning again.

References

1. PAPI: A Portable Interface to Hardware Performance Counters. http://icl.cs.utk.edu/papi, 2005.
2. PathScale EKOPath Compilers. http://www.pathscale.com, 2005.
3. L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, 2004.
4. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 149–159, 1996.
5. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Notices*, 2000.
6. J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS*, pages 340–347, 1997.
7. F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. ACM Workshop on Profile and Feedback Directed Compilation*, 1998. Organized in conjunction with PACT98.
8. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
9. M. Byler, M. Wolfe, J. R. B. Davies, C. Huson, and B. Leasure. Multiple version loops. In *ICPP 1987*, pages 312–318, 2005.
10. H. Chen, J. Lu, W.-C. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture Conference (ACSAC 2004)*, pages 241–255, 2004.
11. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. *ACM Int. Conf on Supercomputing (ICS'05)*, June 2005.
12. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
13. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 23(1), 2002.
14. K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, pages 96–105, 1992.
15. K. D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8:491–523, 1986.
16. P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. PLDI*, pages 71–84, 1997.
17. E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *IEEE PACT 2003*, pages 220–231, 2003.
18. D. Engler. Vcode: a portable, very fast dynamic code generation system. In *Proceedings of PLDI*, 1996.
19. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
20. K. Heydeman, F. Bodin, P. Knijnenburg, and L. Morin. Ufc: a global trade-off strategy for loop unrolling for vliw architectures. In *Proc. CPC*, pages 59–70, 2003.

21. S. Hu, M. Valluri, and L. K. John. Effective adaptive computing environment management via dynamic optimization. In *IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*, 2005.
22. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. Compilers for Parallel Computers (CPC2000)*, pages 35–44, 2000.
23. J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *International Symposium on High Performance Computer Architecture*, 2005.
24. J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *The Journal of Instruction-Level Parallelism*, volume 6, 2004.
25. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proc. AIMSA*, LNCS 2443, pages 41–50, 2002.
26. D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Toward a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *Proc. Int. Conference on Supercomputing*, 2004.
27. E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems*, 2003.
28. R. H. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, 1996.
29. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ACM SIGARCH Computer Architecture News*, pages 165–176, 2004.
30. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
31. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of ASPLOS-X*, 2002.
32. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*. IEEE Computer Society, 2005.
33. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proc. PLDI*, pages 77–90, 2003.
34. S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.
35. X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. PACT*, pages 68–78, 2003.
36. M. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the Symposium on Principles and practices of parallel programming*, 2001.
37. M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Proc. ICPP*, 2000.
38. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998.