Building a Practical Iterative Interactive Compiler

Grigori Fursin and Albert Cohen

ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France
{grigori.fursin,albert.cohen}@inria.fr

Abstract. Current compilers fail to deliver satisfactory levels of performance on modern processors, due to rapidly evolving hardware, fixed and black-box optimization heuristics, simplistic hardware models, inability to fine-tune the application of transformations, and highly dynamic behavior of the system. This analysis suggests to revisit the structure and interactions of optimizing compilers. Building on the empirical knowledge accumulated from previous iterative optimization prototypes, we propose to open the compiler, exposing its control and decision mechanisms to external optimization heuristics. We suggest a simple, practical, and non-intrusive way to modify current compilers, allowing an external tool to access and modify all compiler optimization decisions.

To avoid the pitfall of revealing all the compiler intermediate representation and libraries to a point where it would rigidify the whole internals and stiffen further evolution, we choose to control the *decision* process itself, granting access to the only *high-level features* needed to effectively take a decision. This restriction is compatible with our fine-tuning and fine-grained interaction, and allows to tune programs for best performance, code size, power consumption; we also believe it allows for joint architecture-compiler design-space exploration.By exposing only the decisions that arise from the opportunities suggested by the program syntax and semantics and only when the associated legality checks are satisfied, we dramatically reduce the transformation search space.

We developed an Interactive Compilation Interface (ICI) with different external optimization drivers for the commercial open-source PathScale EKOPath Compiler (derived from Open64); this interface is being ported to the GCC. This toolset led to strong performance improvements on large applications (rather than just kernels) through the iterative, fine-grain customization of compilation strategies at the loop or instruction-level; it also enabled continuous (dynamic) optimization research. We expect that iterative interactive compilers will replace the current multiplicity of non-portable, rigid transformation frameworks with unnecessary duplications of compiler internals. Furthermore, unifying the interface with compiler passes simplifies future compiler developments, where the best optimization strategy is learned automatically and continuously for a given platform, objective function, program or application domain, using statistical or machine learning techniques. It enables life-long, whole-program compilation research, without the overhead of breaking-up the compiler into a set of well-defined compilation components (communicating through persistent intermediate languages), even if such an evolution could be desirable at some point (but much more intrusive). It also opens optimization heuristics to a wide area of iterative search, decision and adaptation schemes and allows optimization knowledge reuse among different programs and architectures for collective optimizations.

1 Introduction

Iterative compilation is a popular approach for optimizing programs for different objective functions on architectures with ever growing complexity, when traditional compilers fails to deliver the best possible performance. Bodin et al. [7] and Kisuki et al. [24] have initially demonstrated that exhaustively searching an optimization parameter space for small kernels can deliver considerable performance improvements in comparison with state-of-the-art, single-run compilers. Cooper et al. [11, 12, 10] and later Kulkarni et al. [25] demonstrated that finding optimal optimization order can also considerably improve code quality and performance. We demonstrated hill-climbing and random iterative search techniques to optimize large applications on a loop-level in [18]. Later Triantafyllis et al. [37, 36] suggested a pruning technique to considerably speed-up optimization heuristic on a fine-grain level inside a compiler. Heydemann et al. [22] used iterative compilation to find trade-off between code size and performance improvement when using loop unrolling and code compression.

Iterative optimization has also been employed in well-known library generators in such systems as ATLAS [38], FFTW [26] and SPIRAL [32] which tune parameters of various transformations to get best performance on a targeted platform. Yotov et al. [39] and Epshteyn [13] use analytical model-based approaches to optimize BLAS libraries.

Many recent results address the iterative tuning of compiler flags, targeting performance or code size for a variety of applications [31, 20, 28–30, 14, 21]. Some of these techniques are already used by companies internally to tune the final settings of their compilers or even available to end-users such as PathOpt tool from the PathScale EKOPath compiler suite [2] that is available since 2004 and allows to find the best combination of flags iteratively using *exhaustive*, *random*, *one of* and *all but one* search methods.

Machine-learning has been also investigated to predict good s transformations and improve hand-tuned compiler heuristics [27, 35, 34, 9, 40, 6]. These works use genetic programming, supervised learning, decision trees, predictive modeling and other similar techniques to tune compiler heuristics usually for one or a few specific transformations.

In our research, we investigate practical aspects of iterative optimizations such as fine-grain tuning of large applications [18, 15], predicting when to stop iterative search [19], run-time optimizations and program low-overhead adaptation at fine-grain level (procedure or loop) for different behaviors [17], investigating the influence of different datasets on iterative search and program performance [16], using machine learning to speed-up optimization process [6] or to speed-up performance prediction for the effective architecture design space exploration [8].

Many interesting transformation and optimization tools have been developed for the purpose of iterative and adaptive compilation. However, most of them are incompatible with each other and not easily portable across architectures. Moreover, using source-to-source optimizations and pragmas can result in heavy, unreadable and non-portable programs that can perform worse on new architectures, so that additional de-optimization/re-optimization techniques may be needed [23]. What makes things worse is an additional, often unpredictable and unquantifiable interference of the tools with hidden/black-box internal compiler optimizations. The most important motivation for our proposed framework is that current tools tend to rewrite and duplicate parts that are currently available inside most compilers, simply because compilers themselves are often seen as untouchable: they rely on intricate transformation engines, undocumented and multi-purpose heuristics, pass orderings and intermediate representations fragile to any modification, providing no support for external tuning and on-demand application of specific transformations. We would like to discrown these myths in this paper and show that current compilers can be used as powerful, flexible yet stable iterative interaction transformation toolsets, their existing heuristics being initially treated as black-boxes, i.e. the inputs and outputs are known but the internal behavior is not, and progressively learned to adapt to a give program on a given architecture.

We developed an Interactive Compilation Interface (ICI) for the commercial PathScale EKOPath Compiler to bias *all* internal optimization decisions and their parameters externally. It is a non-intrusive, stable and flexible way to tune programs at a function, loop or instruction level for best performance, code size, power consumption and any other objective function supported by existing heuristics and an external driver. Current version of the ICI works in the informative and reactive mode, when external tools rather than querying a compiler to apply some specific transformation on a given part of the program, first obtain information from an interactive compiler about all possible legal transformations and their parameters for a specific part of the code and later respond to the compiler to either keep compiler decisions or change them based on statistical and machine learning techniques. This can considerably reduce optimization search space since there is no need to attempt to traverse through illegal or not supported transformations. Still, reactive nature of our method allows external tools to select and parameterize any possible legal sequences of transformations.

We extend the current ICI to support GCC, whose recent versions feature a large number of advanced transformations but with ineffective heuristics. We plan to add support for the reordering of the optimization passes in the GCC to our ICI on a function or loop level, since it already has a relatively clean description of such passes on a global program level. We suggest to substitute current multiple transformation tools with such iterative interactive compilers and use external tools to fine-tune their optimization heuristics. Tools that achieve best results can later be easily and transparently added to the compiler, all the users being immediately able to take advantage of this improvement. Moreover, using unified ICI allows optimization knowledge reuse among different programs and architectures with statistical and machine learning techniques. It also simplifies future compiler development when adding new transformations: e.g., their optimization heuristic can be automatically and continuously learned with machine-learning techniques in the external driver.

2 Motivation

To motivate our research on an open iterative research compiler, we decided to consider some current optimization techniques for `mgrid` application from SPEC CPU2000FP benchmark suite [33]. From [18] we know that source-to-source loop tiling(blocking) and unrolling on `mgrid` from SPEC CPU95FP can reduce its execution time. We used the same source-to-source transformation tool from [18] and hill-climbing search to iteratively find best tiling and unrolling factors for two most-time consuming loops from procedures *resid* and *psinv* for this benchmark on a recent AMD Athlon 64 3700+ plat-

Program version:	Loop from procedure/ transformation:	source-to-source transformation factor:	internal transformation factor:	speedup:
Best variant found with source-to-source transformer	resid/loop tiling	not-found	15 and 182	1.13
	resid/loop unrolling	8	2	
	psinv/loop tiling	not-found	18 and 204	
	psinv/loop unrolling	8	2	
Best variant found with interactive iterative compiler	resid/loop tiling	not-needed	60	1.17
	resid/loop unrolling	not-needed	13	
	psinv/loop tiling	not-needed	9	
	psinv/loop unrolling	not-needed	14	

Table 1. Comparison of best factors found for mgrid benchmark when using source-to-source transformation tool and interactive iterative compiler

form. Later, we applied the same hill-climbing search but using our iterative interactive PathScale EKOPath Compiler.

The results presented in table 1 show that though we reduced execution time using our older source-to-source transformer but interference with internal compiler transformations diminished the result. Previously, we often had to reduce the optimization level of the compiler to remove such ambiguities and sometimes could even improve results, but supporting less transformations than the compiler we could miss some important optimizations. Using interactive iterative compiler, we both avoid this problem and obtain much better result. Moreover, performing optimizations only inside compiler, we reduce and simplify the optimization space since compiler suggests only legal transformations. In addition, many transformations currently applied by the compiler are not profitable (as noticed in [36]) which can also improve the precision of machine learning techniques that we currently use to improve compiler heuristics, quickly find best optimizations or predict best performance, for example (extension of [6, 8]).

Since we bias compiler optimization decision instead of querying it to apply some specific transformation at a particular place in the program, we naturally force compiler to apply aggressively all possible transformations and later de-select unnecessary transformations or change parameters to apply sequences of transformations. Similar method has been used in PathOpt optimization tool (*all but one* search strategy) and later in [29, 30]. The optimization target in these tools are global/procedure-level compiler flags and the main goal is to speed-up the search. However, we noticed, that when optimizing program at fine-grain level in a complex optimization space, turning on all optimizations and later de-selecting some of them would not speed-up the search since multiple ambiguous interactions of various transformations could often considerably degrade the performance. Hence, we use this method to naturally bias compiler optimization decisions externally and we use machine learning techniques to find best optimizations in large optimization spaces quickly (as in [6]) and run-time versioning to further speed-up the search and to adapt to different program behaviors at run-time (as in [17]). We should also note, that some of the source-to-source automatic or manual transformations are still needed since they may be syntactic and difficult to implement inside a compiler. We are currently implementing an ICI for an open-source GCC and

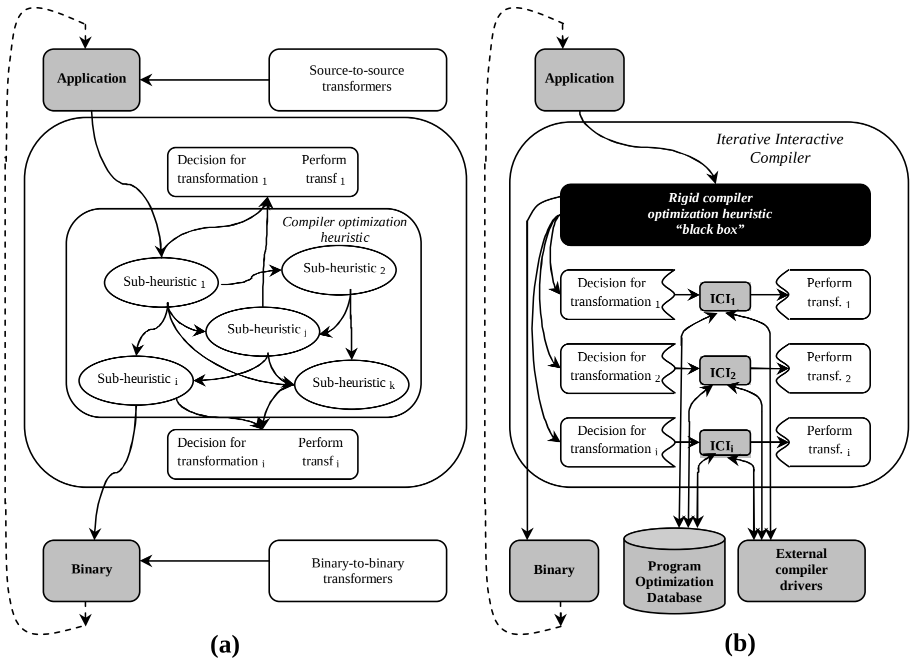
Application
Source-to-source transformers
Decision for transformation 1
Perform transf 1
Compiler optimization heuristic
Sub-heuristic 1
Sub-heuristic 2
Sub-heuristic j
Sub-heuristic i
Sub-heuristic k
Decision for transformation i
Perform transf i
Binary
Binary-to-binary transformers
(a)
Application
Iterative Interactive Compiler
Rigid compiler optimization heuristic "black box"
Decision for transformation 1
ICI1
Perform transf. 1
Decision for transformation 2
ICI2
Perform transf. 2
Decision for transformation i
ICIi
Perform transf. i
Binary
Program Optimization Database
External compiler drivers
(b)

Fig. 1. Internals of (a) current compilers and (b) interactive compilers

plan to gradually add more transformations to this compiler while learning their heuristics automatically.

3 Compiler Framework

Based on our previous experience on iterative optimizations [18, 15, 17, 6, 16], the practical open iterative interactive compiler should have the following features:

- allows simple and unified mechanism to obtain information about all compiler decisions externally and bias them;
- reuses all the compiler program analysis routines to avoid duplications in external optimization tools;
- transparent to user - no project modifications needed;
- removes unnecessary interactions between source-to-source optimizers, compiler and back-end binary-to-binary translators;
- narrows down the optimization search space by using only legal transformations for a given application;
- allows fine-grain (function, loop or instruction level) tuning to get better quality code;
- simplifies compiler development and tuning for new architectures;
- allows reuse of information among different programs and architectures;
- allows modular pluggable third-party transformations and optimization tools.

To address these issues, we suggest the structure of a practical iterative compiler with an Interactive Compilation Interface as shown in Figure 1. This figure depicts an abstract representation of current compilers with hardwired and often ineffective optimization heuristics (Figure 1a) and of the suggested interactive compiler (Figure 1b)). Whenever compiler optimization heuristic makes a potentially ineffective decision to apply some transformation, compiler has to "push out" all the analysis information preceded this decision and provide a user an ability to modify this decision and parameters of this transformation externally through an ICI. We can still treat compiler heuristic as a black box, i.e. where only inputs (compiler decisions) and desired output (performance metrics or other objective function) are known without knowledge about internal structure and transformation interactions, but exposing all decisions at all possible levels (global, function, loop, instruction) allows external tools to automatically learn this behavior and adapt to specific programs and architectures.

The pitfall would be to reveal the compiler intermediate representation and libraries, to a point where it would rigidify the whole internals and stiffen further evolution. To avoid this pitfall, we choose to control the *decision* process itself, granting access the only *high-level features* needed to effectively take a decision. This restriction is compatible with our fine-tuning and fine-grained interaction, and allows to tune programs for best performance, code size, power consumption; we also believe it allows for joint architecture-compiler design-space exploration. By exposing only the decisions that arise from the opportunities suggested by the program syntax and semantics (e.g., detecting that two loops are candidates for tiling), and only when the associated legality checks are satisfied (e.g., checking dependence properties), we dramatically reduce the combinatorial space of program transformation sequences that is searched by external optimization drivers. In fact, we only provide the external optimizer with the combinations currently suggested by the opportunity and legality analyses *triggered on the compilation unit*, granting it access to the only program features embedded into the compiler's specific optimization passes. We believe that this approach of interacting with a compiler will simplify the tuning process of new optimization heuristics and will eventually simplify the whole compiler design where compiler heuristics will be learned automatically, continuously and transparently for a user using statistical and machine learning techniques.

We are developing several communication methods with an interactive compiler - through external file/database, as a client/server connection and through internal calls with a tightly coupled external tool. As a first step, we decided to use external file tor communicate with a compiler similar to common feedback-directed compilation as shown in figure 2. In a *write* mode simply invoked by setting environment variable PATHSCALE_ICI_W to 1, compiler generates an external XML transformation file that contains information about all applied transformations, their parameters and available analysis information. This communication method allows transparent optimizations without any modifications of the source code or project files. An external tool can parse this file with any standard XML parser and modify parameters of existing transformations or disable them. This file is later feeded back to the compiler in the *read* mode by setting environment variable PATHSCALE_ICI_R to 1. In this mode compiler reads and parses modified XML transformation file while optimizing program

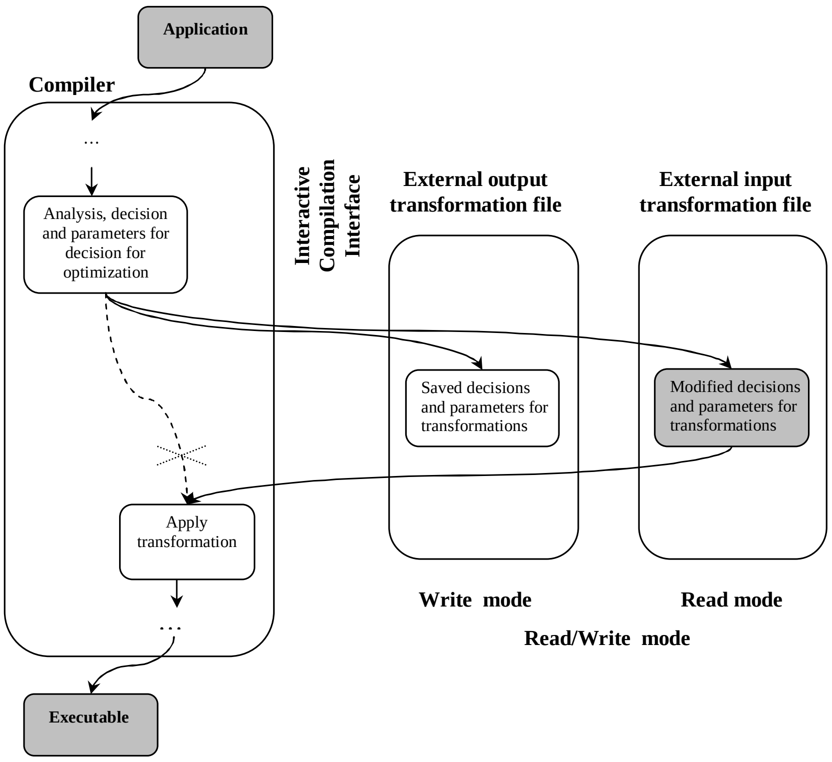
Application
Compiler
...
Analysis, decision and parameters for decision for optimization
Interactive Compilation Interface
External output transformation file
External input transformation file
Saved decisions and parameters for transformations
Modified decisions and parameters for transformations
Apply transformation
...
Write mode
Read mode
Read/Write mode
Executable

Fig. 2. Communication with external tools through transformation file

and substitutes its heuristic decisions and parameters with the matched ones from the transformation file. A sample transformation output for `swim` is shown in figure 3.

When applying transformations that may change loop ordering such as loop interchange, fusion/fission, tiling and others, the subsequent optimization decisions of the compiler can change and will not be matching with the optimization order in the external transformation file. This may result in skipping some externally modified decisions that can cause inconsistencies for external tools when automatically learning the behavior of the program. In such cases, iterative recompilation is required, when interactive compiler and external tool iteratively process the transformation file, refine optimization decisions occurred at each iteration and recompile the program until the the desired sequence of decisions is achieved. To enable such recompilation a *read/write* mode of the interactive compiler is used (when both environment variables PATHSCALE_ICI_W and PATHSCALE_ICI_R are set to 1). In such mode, compiler reads and matches the transformation file with internal optimization decisions, and at the same time produces a new transformation file that contains both modified and unmatched optimization decisions. The iterative recompilation algorithm is shown in figure 4. Naturally, this mode is also used to apply any legal sequences of transformations, thus demonstrating how a

```
<?xml version="1.0"?>
<compiler_ici>
 <file_name="swim.f">

  <transformation name="unroll_and_peel">
   <function>calc1</function>
   <loop_number>4</loop_number>
   <depth>1</depth>
   <decision>4</decision>
   <factor>7</factor>
  </transformation>

  <transformation name="unroll_and_peel">
   <function>calc1</function>
   <loop_number>3</loop_number>
   <depth>1</depth>
   <decision>4</decision>
   <factor>7</factor>
  </transformation>
  …

</file_name>
</compiler_ici>
```

Fig. 3. Example of the transformation XML file for `swim`

```
    clear transformation_file_out.xml
    set PATHSCALE_ICI_W to 1
    compile program
        (write transformation_file_out.xml)
    set PATHSCALE_ICI_R to 1
_label_recompile:
    copy transformation_file_out.xml to
        transformation_file_in.xml
    modify transformation_file_in.xml if needed
    compile program
        (read transformation_file_in.xml,
        write transformation_file_out.xml)
    if transformation_file_in.xml not the same
        as transformation_file_out.xml
        go to _label_recompile
```

Fig. 4. Iterative recompilation algorithm to apply sequences of transformations

compiler with a hardwired heuristic can become a flexible transformation tool with our Interactive Compilation Interface. However, for some large applications, using external file and *read/write* compiler mode for interaction with external tools may require several recompilation and can be time consuming. This motivated us to develop a prototype of a client/server communication method where decisions can be modified during during compilation time and therefore no further recompilation is needed.

Currently, we added support to modify internal PathScale EKOPath compiler optimization decisions for the following transformations:
inlining, array padding (global/local), loop fusion/fission,
loop interchange, loop blocking, loop unrolling, register tiling,
prefetching.

4 Tools and Experiments

We expect to substitute multiple often non-portable and non-compatible transformation tools with our iterative interactive compiler as shown in figure 5. In this case, optimizations will be performed continuously and transparently to user, i.e. it will not require any modifications of a source code or project files. All information about best found optimizations on a given architecture is saved in a *Program Transformation Database* kept along with a program. This simplify application development, optimization and portability since no information about optimizations is now hardwired in the source code of the program and there is no dependence on multiple external tools that may not be available on some architectures. Moreover, *Program Transformation Database* keeps information about all best possible optimizations for different program behaviors on different architectures (as described in [17]). Therefore, whenever program is ported to a new platform, the optimization process can start from already found best configurations from multiple programs thus reusing optimization knowledge among different

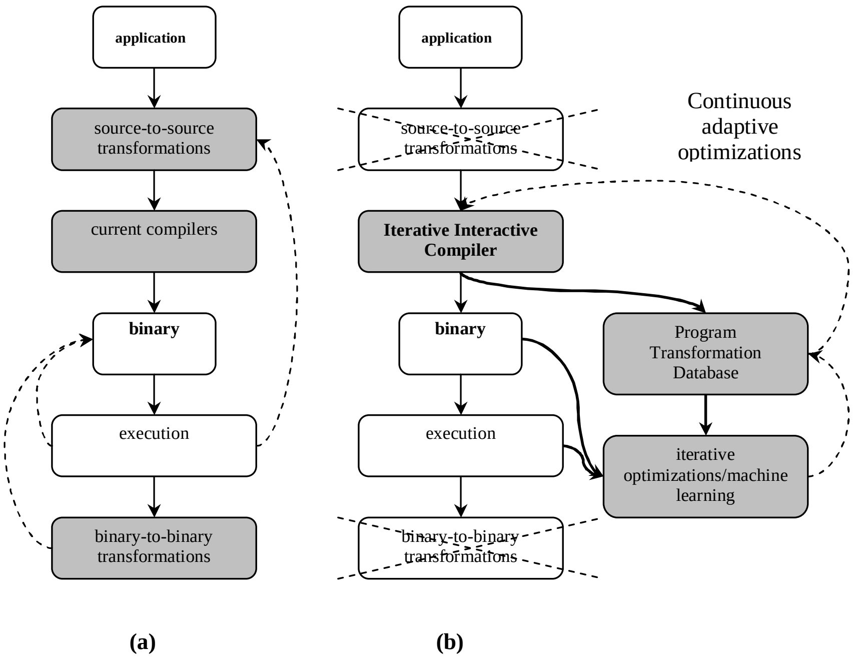
application
source-to-source transformations
current compilers
binary
execution
binary-to-binary transformations
(a)
application
source-to-source transformations
Iterative Interactive Compiler
binary
execution
binary-to-binary transformations
Continuous adaptive optimizations
Program Transformation Database
iterative optimizations/machine learning
(b)

Fig. 5. Iterative optimization scenario using iterative interactive compiler

programs and architectures, behaving as a collective compilation system and considerably narrowing down the optimization search space.

Since 2004, we and our colleagues actively used our iterative interactive compiler in different projects and developed or prototyped the following support tools and optimization drivers.

- Continuous iterative optimization driver with run-time adaptation at function, loop-level or instruction level using low-overhead phase detection technique (as described in [17]. We use *exhaustive*, *random* and *hill-climbing* search strategies (as in [7, 18, 15]). We also use a *all but one* search strategy on a fine-grain level similar to the one implemented in the PathOpt tool from the original PathScale EKOPath compiler distribution where global compiler flags are turned all on at the first step and later turned off one by one.
- Driver to continuously collect all possible optimization parameters. This driver is useful when compiler optimization heuristics is treated as a black box and its behavior is learned automatically to collect all varieties of optimization decisions and parameters automatically and transparently to a user, instead of listing them separately and keeping them up-to-date.
- Driver to automatically and continuously rebuild compiler optimization heuristic, and adapt to a specific architecture using statistical methods and collective optimization knowledge reuse among different programs and architectures.

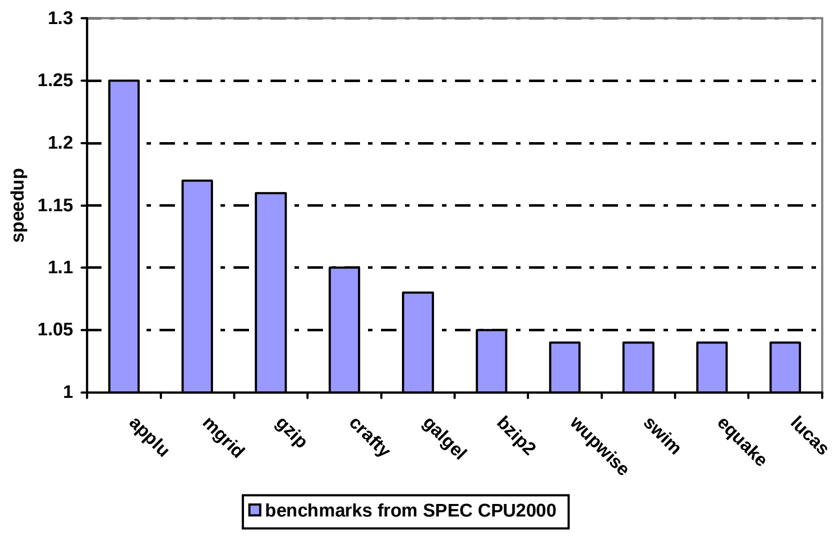

Fig. 6. Speedups of several SPEC CPU2000 applications in comparison with -Ofast optimization level when using interactive PathScale EKOPath compiler with hill-climbing search

– Prototype framework to replace a model-based compiler heuristic with automatically learned one. We use our iterative interactive compiler together with the WEKA tool [3], which is an open-source machine learning software package. Our preliminary results target loop interchange; we will revisit the main optimizations for which machine learning techniques have been proposed so far, as well as experiment with more challenging ones.

We developed multiple support tools and external optimization drivers for our iterative interactive compiler. We already created various tools to support our ICI-enabled compiler and we developed several external optimization tools that uses iterative search to find best transformations and their parameters to minimize execution time of programs. We use *exhaustive*, *random* and *hill-climbing* search (as in [7, 18, 15]), *all but one of* search (implemented in the PathOpt tool from the original PathScale EKOPath compiler suite [2] when all compiler flags are turned on and later turned-off one by one). We also use this interactive compiler with program optimizations at fine-grain level for run-time program adaptation described in [17].

Since the purpose of this article is mainly to describe the building of an interactive iterative compiler, we decided to leave complex iterative optimization schemes for the journal version of the paper and selected a relatively simple hill-climbing optimization scheme as described in [18, 15]. We performed all experiments on AMD Athlon 64 3700+ at 2.4GHz, with an L1 cache of 64KB and an L2 cache of 1MB, and 3GB of memory; the O/S is Mandriva Linux 2006. We instrumented the open-source commercial PathScale EKOPath Compiler 2.x [2] to enable Interactive Compilation Interface to allow external tuning of its optimization heuristic. It has a mature but ambiguous optimizer with many transformations available, based on the ORC compiler, and is specifically tuned to AMD processors. We selected several programs from the SPEC 2000 suite [33] and ran our search tool continuously and transparently to user until all transformations and their parameters have been analyzed. Whenever possible, we used run-time versioning scheme from [17] to considerably speed-up iterative search and allow further run-time adaptation. We needed from around 500 to 5000 runs (with 16 versions of examined functions during one run) per program to finish optimizations. The speedups shown in figure 6 in comparison with the best -Ofast optimization level of the EKOPath compiler demonstrate that it is possible to beat the state-of-the-art compiler even on large programs with the most aggressive optimization level enabled using simple Interactive Compilation Interface and external iterative optimization drivers. We will describe all other optimization scenarios in more detail in the journal version of the paper and will make all the software, source codes and data publicly available at [4].

5 Conclusions and Future Work

In this article we demonstrated a simple, practical and non-intrusive way to turn current rigid compilers into powerful interactive transformation toolset with an Interactive Compilation Interface that allows to bias compiler optimization decisions externally. We show how to avoid the pitfalls of rigidifying the compiler internals, while granting access to rich-enough features to take performance-critical decisions. We assist the external optimization tools in considerably reducing the size of the optimization search space by analyzing only possible transformations, and in continuously collecting the most interesting sets of transformation parameters. We developed an ICI for the commercial open-source PathScale EKOPath Compiler and, within last 2 years, developed different support tools to optimize programs at loop or instruction level continuously and transparently to a user. We use it to automatically optimize programs for the best performance, code size, power consumption and hardware designs. We plan to make all the software publicly available at [4].

Based on this work, we are currently developing a unified extensible and portable ICI in the latest version of GCC [5] with a support from IBM, Philips (NXP), STMicro, ARC and multiple universities within HiPEAC network of excellence [1]. We enable an access to the most influential compiler transformations (including OpenMP directives) with ineffective optimization heuristics and enable optimization pass reordering at a function or loop level. We are working on the optimization naming conventions to enable portability and automatic knowledge reuse using machine learning between different compilers and their versions. We plan to add ICI to the JIT-compilers (Jikes, .NET compilers) to unify the run-time optimizations as well. One of the most advantages of a unified ICI is that it enables life-long, whole-program compilation research with collective reuse of the knowledge (program features, analysis results and transformation decisions) across different programs and architectures without the overhead of breaking-up the compiler into a set of well-defined compilation components (communicating through persistent intermediate languages), even if such an evolution could be desirable at some point (but much more intrusive).

We are using our toolset in the EU-funded MilePost, SARC and GGCC projects. Within MilePost, we aim at dramatically changing and simplifying the design of future compilers on rapidly evolving hardware by automatically and continuously learning the best optimization settings for a given program, context, platform and any given set of compiler transformations. Within SARC, we facilitate collective optimization-space exploration of the architecture and compiler, on a heterogeneous chip multi-processor. Within GGCC, we contribute to the emergence of a production-quality standard for whole-program analysis and optimization. We believe this is a major research and development direction towards a practical and general-purpose development toolset based on integrative compilation.

6 Acknowledgments

The authors would like to thank Olivier Temam and Michael O'Boyle for the helpful discussions and support throughout the project, and anonymous reviewers for a valuable feedback. This research is partly funded by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

1. European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). `http://www.hipeac.net`.
2. PathScale EKOPath Compilers. `http://www.pathscale.com`.
3. WEKA: Open-source machine learning software. `http://www.cs.waikato.ac.nz/ml/weka`.
4. Interactive Compilation Interface for PathScale EKOPath Compiler. `http://sourceforge.net/projects/pathscale-ici`, 2004.
5. GCC Interactive Compilation Interface. `http://sourceforge.net/projects/gcc-ici`, 2006.
6. F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.
7. F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.
8. J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006)*, October 2006.
9. J. Cavazos and J. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.
10. K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
11. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
12. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.
13. A. Epshteyn, M. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Hawthorne, NY, USA, 2005.

14. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
15. G. Fursin. *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, University of Edinburgh, United Kingdom, 2004.
16. G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007) (to appear)*, January 2007.
17. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.
18. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
19. G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate method for determining a lower bound on execution time. *Concurrency: Practice and Experience*, 16(2-3):271–292, 2004.
20. M. Haneda, P. Knijnenburg, and H. Wijshoff. Generating new general compiler optimization settings. In *Proceedings of the 19th annual international conference on Supercomputing (ICS'05)*, pages 161–168, New York, NY, USA, 2005.
21. M. Haneda, P. Knijnenburg, and H. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, 2006.
22. K. Heydemann and F. Bodin. Iterative compilation for two antagonistic criteria: Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.
23. S. Hines, P. Kulkarni, D. Whalley, and J. Davidson. Using de-optimization to re-optimize code. In *Proceedings of the EMSOFT Conference*, pages 114–123, 2005.
24. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC2000)*, pages 35–44, 2000.
25. P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.
26. F. Matteo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.
27. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.
28. Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *Proceedings of the International Conference on Supercomputing*, 2004.
29. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.
30. Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, Seattle, WA, September 2006. IEEE Computer Society.

31. R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff. Statistical selection of compiler options. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 494–501, 2004.
32. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.
33. The Standard Performance Evaluation Corporation. `http://www.specbench.org`.
34. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.
35. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.
36. S. Triantafyllis, M. Vachharajani, and D. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.
37. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.
38. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998.
39. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 63–76, 2003.
40. M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 317–327, 2005.