Hybrid Performance Analysis to Accelerate Compiler Optimization Space Exploration for In-Order Processors

Nicolas Inglart*, Smail Niar*, Albert Cohen†

*LAMIH, University of Valenciennes, France
†INRIA-FUTURS, ALCHEMY Project, Orsay, France
{nicolas.inglart, smail.niar}@univ-valenciennes.fr, Albert.Cohen@inria.fr

Abstract. We investigate the problem of finding the most adequate compiler optimization options to compile a given application to an in-order processor architecture. This is a well known problem, but accuracy (effectiveness) and speed of the decision remain challenging. We present a two-phase method for a rapid performance evaluation of a set of compilation optimizations. The method combines dynamic and static code analyzes and thus allows benefiting from the accuracy of the former and the speed of the latest. This approach makes our method interesting for Compiler Option Space Exploration (COSE).

1 Introduction

Semiconductors progress according to Moore’s law, through the reduction of the scale of individual components, and through the growth of hardware resources. Unfortunately, the complexity of processor (micro-)architecture increases accordingly, and the rise of multi-core processors is the latest trend in this direction. The complex interplay between architecture components challenges traditional compilers. Static optimizations lack the profitability metrics to accurately model the performance impact of program transformations. To complement analytical models in static compilers, there is a growing need for a software tool chain to efficiently utilize the available hardware resources. Iterative, feedback-directed optimization is the main approach to design such a tool chain. This paper combines dynamic and static performance estimation to accelerate iterative optimization. We focus on the estimation of single-thread performance on in-order processors, for three main reasons.

1. In-order pipelines are more predictable in general thus taking advantage of any static performance analysis.
2. The trend in multi-core processor design is to increase the density of computational engines and to lower power consumption; this trend favors in-order execution, while deep and wide pipelines are still important to exploit fine-grain parallelism.
3. In-order micro-architectures— and VLIW processors in particular — greatly depend upon the compiler to achieve good performance.
Generally, performance can be improved at two levels:

– At the hardware level, where the optimization consists in choosing the best micro-architectural configuration. To this end, Design Space Exploration [21, 15] or DSE is generally conducted to find the most adapted functional unit configurations (cache sizes, branch predictor architecture, etc.). This approach is widely used in the embedded system area. However, it is only applicable if the architecture is in the design phase or if the hardware is reconfigurable [19].

– At the software level, where the optimization consists in producing a binary code that makes the best usage of the existing hardware resources. This second method is less constraining than the first one because it is micro-architectural and application independent. When the compiler optimization passes are fixed, this approach consists in finding the best compilation options for the target architecture. This is the aim of Compiler Option Space Exploration (COSE) [20]. In the particular case of Itanium and VLIW architectures, compile-time optimization is utterly important: in this class of micro-architectures, fine-grain parallelism is explicit and results from scheduling and mapping decisions in the back-end compiler.

In this paper, we investigate COSE for the case of the Itanium architecture, and we present a new performance evaluator to speedup the COSE process.

Any increase in processor complexity and in the expressive power of program optimization techniques tends to result in an increase of the size of the optimization space, and a significant disparity in the performance of the different compilation alternatives. These alternatives are called configurations throughout this paper. In addition, interaction between the different optimizations could, in some cases, diminish performance. Unfortunately, it is not always possible to determine the best optimization sequence a priori, without taking source code specificities into account [20].

In this paper, we propose a performance evaluator to estimate the each configuration for a COSE. Using this evaluator, a high performance configuration can be determined in a relatively short time period. As shown in Figure 1, our COSE tool chain has three components including the configuration performance evaluator. Compared to existing work in the domain, our COSE evaluator has two original features.

1. First, a hybrid static/dynamic approach was adopted for the evaluator, which greatly improves the accuracy of static models while significantly reducing the combined compilation/execution time of feedback-directed dynamic evaluators. We believe this hybrid approach contributes to the viability of iterative compilation for future and emerging processors.

2. Second, the evaluator integrates multiple specially-designed models of micro-architecture components: currently, it includes branch prediction, dependences, fine-grain parallelism and instruction cache. We also propose two original, differential cache models to compare optimization configurations. Except for the cache models whose empirical evaluation is not completed,
we show that our model closely match the effective influence of a given optimization on the execution time.

2 Related work

Cose is motivated by the increasing number and aggressiveness of compiler optimizations, combined with the parameters and switches to control their application. The size and complexity of the optimization space in modern compilers results in important disparity between the performances of different optimization configurations [20]. COSE can be accelerated in two different ways. The first attempts to reduce the number of explored configurations, whereas the second aims to accelerate the performance estimation for a given configuration.

Many different heuristics [1, 20, 2, 16] have been proposed to accelerate COSE by reducing the number of configurations. E.g., Triantafyllis et al. [20] recommended pruning the configurations by exploiting feedback from previously evaluated configurations. Agakov et al. showed that machine learning, and predictive modeling in particular, can effectively narrow huge search spaces to a handful of configurations [2]. However, as the complexity of the search space increases, due to the aggressiveness of the optimization, or when considering larger benchmarks, recent results tend to confirm that the exploration of larger sets of candidate configurations will be necessary [4].

As an alternative to search-space narrowing approaches, several authors have sought to avoid the bottlenecks caused by performance estimation itself, for example, relying on hardware counters [11] that allow performance values to be read directly on the processor, with very low overhead. The tool proposed by Triantafyllis et al. [20] uses a static performance estimator to compare performance of different configurations. However, though using static analysis can provide rapid performance estimations, it tends to lack accuracy. Others projects [7] has
focused only on dynamic analysis and the phase behavior of programs. Cava-\nzos et al. combine search space narrowing and dynamic analysis using hardware \ncounters [4].

Cooper et al. [5] propose, through their ACME framework, the use of virtual \nexecution. In their approach, one application execution is carried out to build a \ncontrol flow graph annotated with basic block execution occurrences. Following \na set of rules, this graph is modified according to the chosen code transformat-
\ions. The major drawback of this approach is the inaccuracy of the obtained \nannotations, resulting in some error to find the best optimizations. To overcome \nthis problem, Kulkarni et al. [14] enhance the previous approach by allowing \ndynamic execution of a small set of functions within the application. Even if the \ninstruction count accuracy has been ameliorated, the exploration time increases. \nFinally, in the two previous works, the only criterion used in determining the \nbest transformation is the instruction count. Whereas in embedded systems this \ncriterion could be sufficient, in the case of high performance processors, such as \nEPIC/Itanium, this metric alone is not adequate.

Our hybrid method is the first to integrate dynamic analysis to refine the \naccuracy of static estimations. It can be combined with search space narrowing \nfor further acceleration.

3 Combining static and dynamic profiling for COSE

The method we propose for performance evaluation is based on four observations.

1. Performance estimation accuracy in COSE can be improved using dynamic \nstatistics gathered across the execution of the entire application.
2. Some of these execution statistics (also called global statistics) are consistent \nfor a wide range of configurations. E.g., when the order of memory accesses \nis not modified, the cache miss/hit statistics are not changed.\footnote{Except on \nshared instruction-data caches, but this effect can be neglected in practice.}
3. Even when execution statistics differ from one configuration to another, it \nmay be possible to infer a correction of the statistical data from the analysis \nof static differences between these configurations. E.g., when the order of \nmemory accesses is not modified, we will show that the latency between a \ngiven load and the first use (which may block the pipeline) can be adjusted \nfrom the difference in cycle count in both configurations.
4. The correspondence between a set of instructions in the binary code and \nthe instruction in the source code to which they correspond can easily be \nobtained. Debug information is of great help, when it is carried over compiler \noptimizations, which is generally the case in modern compilers.

These observations allowed us to design a new configuration performance eval-
uation method. As shown in Figure 2, the method has two phases.

The goal of the hybrid evaluation is to compute CPU\_CYCLES, the total \ndynamic cycle count for the application.
Fig. 2. Performance evaluator details
3.1 Dynamic analysis phase

In the first phase, dynamic features of source code are collected. The aim is to gather global execution statistics once and for all, then to reuse these statistics across the configurations to be explored in the second phase (through static analysis). Global statistics are collected on binary code, but attached to the source code; they depend on the input dataset of the application. To obtain these global statistics, source code instrumentation is performed. The most important statistic is the execution frequency of each instruction in the source code. This value is used throughout the static analysis.

The first phase generates a trace of the source code instructions executed, and the stall latency of every memory instruction. The stall latencies can be collected through hardware counter sampling [11]. These latencies will allow to adjust the effective stall cycles through the static analysis of any configuration. The relatively high cost of this first phase is compensated by the fact that only one execution is needed for the full optimization space exploration.

3.2 Static analysis phase

The second phase is further divided into two steps and is completed for each configuration. The first step links the source code to the new binary code of the configuration, by tagging each instruction in the binary code with the corresponding instruction from the source code. The second step consists of obtaining static statistics for the binary code and combining them with global statistics gathered in the dynamic analysis phase. These static statistics are much faster to collect than executing the new configuration; they are combined with global statistics to obtain more accurate performance estimations.

The hybrid evaluation of the execution time relies on the following breakdown of the cycle count:

\[
\text{CPU}_\text{CYCLES} = \text{WORK} + \text{BRANCH} + \text{ICACHE} + \text{DCACHE}.
\]

WORK is the number of cycles to execute the application, assuming a perfect branch predictor and perfect instruction and data caches. Consequently, the value of WORK is only dependent on the degree of the ILP and on the latencies between dependent instructions in the application. In-order processors — and the Itanium architecture in particular — facilitates the estimation of WORK, using a simple resource model and an instruction scheduler, or simply relying on the instruction bundle format of a VLIW processor. On the Itanium processor, the Intel compiler reports static cycle count estimates based on stop bits and latencies [18, 13]). This information can be processed automatically, as previously demonstrated in the MAQAO project [6].

BRANCH is the penalty due to branch miss-predictions.\(^2\) For each branch, we rely on the back-end compiler to provide branch probabilities. Intel and GNU

\(^2\) We do not consider indirect branches and address prediction at the moment.
compilers [13, 8] support a wide range of dynamic and static branch outcome analyzes; they report the results as comments in the assembly code or through separate dumps. Although compilers accurately predict branch outcomes through profile-directed compilation, static predictions are often unsatisfactory outside of static control loop nests. Fortunately, optimizing compilers support a variety of statistical methods to maintain and infer profile information across program transformations; a good example is the application of the Dempster-Shaffer theory to maintain (dynamic or static) profile information across control-flow restructuring [12]. We believe a similar approach can be taken to refine the precision of branch probabilities from the original program to any given configuration; this study is left for future work. Then, to estimate misprediction penalty, we assume the most frequent case (taken or not taken) is always predicted, and the other case is always mispredicted.

Finally, ICACHE and DCACHE correspond to the cycle penalty due to instruction and data cache misses, respectively. These two values are first estimated from the execution trace obtained during the dynamic analysis phase, then adjusted according to the differences between the original program and the configuration considered. The following sections outline the motivation and design of these differential estimators.

3.3 Data access latency

The estimation of the latencies due to data cache misses is one of the key issues to assess the performance of a given configuration. We are not aware of any analytical cache model that is precise enough to cover a wide variety of transformations, and that is not restricted to static control loop nests [9].

The method we propose adjusts the latencies computed in the first dynamic phase using static analysis of the instruction displacement in the new configurations. To avoid resorting to analytical cache modeling, our method currently assumes that the order of data accesses is not affected by the compiler optimizations. Although surprising at first glance, this assumption is quite relevant given the state-of-the-art in production compiler optimizations. Indeed, locality enhancing-transformations have been proposed for loop nests, and may achieve strong speedups on a variety of kernels [3]; yet those transformations are largely out of reach of production compilers, except on a few benchmarks for which the optimizations have specially been trained [17]. There are many reasons for the failure of current compilers to significantly alter the order of memory accesses, including the inaccuracy of pointer and dependence analysis, restrictions on the class of programs amenable to such transformations, and the fragility of complex transformation sequences for locality enhancement [17, 10]. In addition, many optimizations are able to eliminate memory accesses (scalar promotion) or introduce new ones (register spilling). The former are not a problem for static performance modeling, while the latter can reasonably be considered as cache hits (due to the relatively small number of such accesses in general). Overall, the preservation of memory access ordering is well supported by our empirical studies.
Based on this assumption, a list of stall times is obtained using the instruction trace obtained in the dynamic analysis phase. Then, during the static analysis phase, we compute the difference in cycles between each memory instruction and the first following data dependent instruction. This step can be easily integrated in conjunction with the computation of the static cycles count (WORK). The penalty due to the $j^{th}$ access of memory instruction $I$ is derived as follows:

$$DCACHE_{STALL}(I, j) = \max(0, DCACHE_{DELAY}(I, j) - DIFF(I)),$$

where $DCACHE_{DELAY}(I, j)$ represents the data access delay of the $j^{th}$ access of memory instruction $I$, as computed through the dynamic analysis phase, and $DIFF(I)$ represents the distance, in number of cycles, between the memory instruction $I$ and the first instruction that uses the data (relying on the computation of static cycle count in Section 3.2).

### 3.4 Instruction access latency

The computation of instruction fetch latencies relies on the assumption that basic block layout does not widely change across configurations. A priori, this assumption reduces the accuracy of our method in presence of code layout optimizations. Such optimizations are commonly supported by production compilers, yet only combinations of optimizations resulting in widely different layouts may yield coarse approximations; we believe this situation is not the common case, but further studies will be required to validate the assumption.

The evaluation is divided into two steps. The first step identifies which instructions produce misses, whereas the second step evaluates the latency for each of those instructions. The first step starts from the trace generated in the dynamic analysis phase: it considers the most frequently executed paths from the compilation reports of the configuration, retrieve the addresses of basic blocks on these paths (from the trace), and simulate the instruction cache on each path. The second step relies on the miss latencies collected during the first phase, assuming these latencies do not change much over all configurations (for a given basic block). Several issues remain to be investigated, including the tuning of the path construction and selection heuristic, block-level comparison across configurations (more complex than instruction-level), and more precise miss latency estimation.

### 4 Results

We present experimental results to demonstrate the potential of our method for accelerating configuration performance estimation in COSE. We applied our evaluator to several benchmarks taken from the SPEC CPU2000 and Media-bench suites.

Due to technical problems in hardware counter sampling for cache miss latencies, and due to the incomplete state of the instruction cache model, we prefer
to report results for a simplified version of our hybrid method. Technically, we ignore the DCACHE and ICACHE terms in the computation of CPU_CYCLES. Of course, this tends to systematically underestimate the cycle count, which will be confirmed in our experiments.

Figure 3 compares the execution time (in cycles) for a COSE as measured by the hardware counters (HC) of the Itanium2 processor, with the value estimated by our hybrid method. This figure corresponds to the bitcount benchmark. The behavior of our estimator for the other benchmarks is reported in Figures 4, 5, 6 and 7. The configurations in the horizontal axis were sorted by increasing number of cycle count, as recorded by HC. Although there is a significant gap between the HC curve and the curve representing our method, the best solutions for the two methods are very close and the two curves shown are parallel. We expect that the gap will be reduced when integrating the hybrid estimation of cache stall cycles.

Figure 8 shows the ability of our evaluator to find the best configuration. For each benchmark, the first graph (ERROR) represents the gap between the real execution time of the best configuration found by HC and the execution time of the best configuration determined by our method — without cache penalty. The second graph (AVERAGE GAP) represents the gap between the execution time of the best configuration and the average execution time over all configurations with HC. The third graph (MAXIMUM GAP) represents the gap between respectively the execution time of the best configuration and the execution time of the worst configuration. Both of these configurations are determined by the HC. The error on this last graph is below 2% for all benchmarks. Compared to the variance and average of configuration performances, this value is negligible.

Fig. 3. Real vs. estimated execution time (in cycles) for bitcount
Fig. 4. Real vs. estimated execution time (in cycles) for facerec

Fig. 5. Real vs. estimated execution time (in cycles) for fft
Fig. 6. Real vs. estimated execution time (in cycles) for gzip2

Fig. 7. Real vs. estimated execution time (in cycles) for sha
5 Conclusion and future work

This paper describes an original two-phase hybrid method to estimate configuration performance for Compilation Optimization Space Exploration (COSE). It applies to in-order processor pipelines, and combine the advantages of the existing static and dynamic approaches. The method was evaluated on the Itanium2 processor. As different models (e.g., caches, branch predictor, producer/consumer instruction latencies, etc.) have been integrated into the evaluator, the configuration performance estimator can easily be adapted to other processor architectures. Cache miss modeling was discussed, and a differential analysis combining dynamic and static phases seem possible; this extension is still under investigation, but we expect these models will allow further improvements in accuracy, and to extend the estimation to a larger spectrum of aggressive optimizations. Our performance estimation is very fast and relatively accurate: the best configuration determined by our method is at most 2% slower than the real best configuration for the tested benchmarks.

References


