Combining Experimental Search and Performance Model for Adaptive Optimization

Julien Jaeger[1] and Denis Barthou[1,2]

[1] University of Versailles St Quentin, France
[2] INRIA Saclay, France

Abstract. The increasing complexity of hardware features in modern processors makes compilation for high performance very challenging. Performance models used by compilers are too simple to take into account this complexity and choose accordingly the most effective optimization sequence.
Adaptive compilation is now a widespread approach relying on an exploration for optimization sequences or compiler flags and on code execution in order to evaluate precisely performance. The main drawback of this approach is its very high cost, that is partially addressed by efficient search techniques based on genetic or machine learning algorithms.
This paper presents a novel approach for adaptive compilation, relying on performance evaluation of only fragments of the code, named constant performance codelets, and on a simple performance model. The search for transformations leading to these codelets is user-defined through pragmas. We show on three large applications (two numerical simulations and a genomic application) that the performance prediction for the optimized function is quite accurate and that substantial speed-up can be reached on Itanium2 architecture.

1 Introduction

The increasing complexity of hardware features in modern processors makes compilation for high performance very challenging. Effects of one single optimization depend on both the application and the target architecture and is hard to predict. Static compilers resort to simplified performance models, focusing on one metric (such as cache misses for instance) assessing the assumed effectiveness of the optimization. The combined effects of a sequence of optimizations is therefore most often unpredictable and prevents compilers from finding most effective optimization sequences. Actually, the risk of using an approximate performance model is as much to degrade performance as to miss optimization opportunities. This leads most production compilers to trigger some optimizations only in very favorable cases (xlc compiler and vectorization for BlueGene for instance [16]).

One approach to deal with the problem of simplistic performance model is to rely on performance measures, making code execution part of the compilation time. Adaptive compilation drives selection of optimization sequences by feedback performance evaluation [20]. To limit the search among possible optimization sequences, several techniques have been proposed[21]: adaptive compilers

resort to genetic algorithms[6], machine learning algorithms[2] and some amount of enumeration. Auto-tuning libraries also use adaptive compilation techniques (Spiral[30], FFTW[13], ATLAS[36], Stapl[28] for instance) and have shown their effectiveness even compared to hand-tuned codes. With the exception of BLAS library where it has been shown that a pure performance model approach for ATLAS can be competitive with the empirical search version [39], the main drawback of iterative compilation is its large compilation time. As for each change of compiler flag, optimization parameter or sequence, the whole code is executed. Most of the research on iterative compilation has focused on limiting the search, and consecutively, the total execution time (see [14] for instance, based on the different phases of an execution). However, these approaches assume some properties on dynamic execution (the input is representable, the run of the test is repeatable) whereas our method is based only on static decomposition of the code based on control (and dataflow) properties.

This paper proposes a novel approach for adaptive compilation, combining empirical search to a performance model. The main contribution is to describe a method that searches for many code variants and predicts performance accurately while only small code fragments are executed. The method is threefold: (i) source to source transformation sequences are explored according to user-defined pragmas, generating as many code versions, (ii) for each version, some inner loops, named codelets, are taken out of the application and executed, (iii) performance evaluation of each code version is obtained by combining codelet timing measures and a simple performance model. This approach is an extension of the method presented in [3] for library generation of linear algebra codes, and we generalize it so that real applications can be targeted. We show with experimental results on three applications (lattice QCD simulation, molecular simulation and a genomic application) that our performance model is very accurate and that substantial speed-up can be reached for sequential executions on Itanium2 architecture.

The paper is organized as follows. The rest of the introduction presents some motivating example and recalls the framework of meta-compilation that we use. Section 2 defines the codelets the search tries to discover and the properties expected from these codes. Section 3 describes more in depth the search methodology we used and Section 4 presents how performance of the whole code is inferred from previous codelet evaluation and blocks of copy/prefetch evaluation. Section 5 shows the results obtained on the three applications and we conclude with related works and conclusion remarks.

1.1 Motivating Example

Consider the example in Figure 1.a, taken from the hot spot function in GIBBS application developed by U.of Orsay and French Oil Institute (IFP). GIBBS ensemble simulation (see [31] for instance) is a molecular simulation computing fluid phase equilibria properties using a Monte-Carlo method.

The code has a triangular loop domain (and bounds are not statically known), indirect accesses in statements 3,4,9 and 10 and dynamic control due to condi-

tionals. Note that for presentation details, some statements, similar to 4 and 5, have been omitted. The code does not look as a very good candidate for compiler optimizations but the application context ensures that the working set of this code is and fits into the L2 cache of an Itanium2. Let us show that a realistic performance evaluation of this code, depending on the values of loop bounds `molstart` and `molend` can be computed without executing the whole code.

Applying for instance the following transformations lead to the inner loops presented in Figure 1.b: Statements 14 and 15 are reductions and are responsible of the only loop-carried dependences. A 1D tiling of loop j is possible, at the expense of tail code with triangular iteration domain due to the initial triangular iteration domain. Fissioning the resulting inner loop into three groups of statements $3-6$, $7-13$ and $14-15$ produces the three loops presented in Figure 1.b.

```
  for (i=molstart; i<molend; i++)
1 |tk = TypeLJ[S[i]]
2 | k = S[i]
  | for (j=i+1; j<=molend; j++)
3 | | tl = TypeLJ[S[j]]
4 | | rijx = X[k] - X[S[j]];
5 | | rijx1 = rijx - boxl1 * round(rijx / boxl1);
6 | | rij2 = rijx1 * rijx1;
  | | if (rij2 < mindist)
7 | | | energy = 2 * infinity;
  | | else
8 | | | energy=0;
  | | | if (rij2 < cutoff2)
9 | | | | epsilon = Eps[ncftype * tl + tk];
10| | | | sigma = Sig2[ncftype * tl + tk];
11| | | | rij2 = sigma / rij2;
12| | | | rij6 = rij2 * rij2 * rij2;
13| | | | energy = epsilon * (rij6 - rij6 * rij6);
14| | Energy_change[j]= Energy_change[j] + energy;
15| | Energy_change[i]= Energy_change[i] + energy;
```

(a) Initial version of GIBBS main function. Some statements have been omitted out of simplication.

```
codelet1(X,S,TypeLJ,boxl1,k,Rij2)
  for (jj=0; jj<STEP; jj++)
3 | tl = TypeLJ[S'[jj]]
4 | x = X[k] - X[S'[jj]];
5 | x1 = x - boxl1 * round(x / boxl1);
6 | Rij2[jj] = x1 * x1;

codelet2(Rij2,tl,tk,Eps,Sig2,E)
  for (jj=0; jj<STEP; jj++)
  | if (Rij2[jj] < mindist)
7 | | E[jj] = 2 * infinity;
  | else
8 | | E[jj]=0;
  | | if (Rij2[jj] < cutoff2)
9 | | | e = Eps[ncftype * tl + tk];
10| | | s = Sig2[ncftype * tl + tk];
11| | | r3 = s / Rij2[jj];
12| | | r6 = r3 * r3 * r3;
13| | | E[jj] = e * (r6 - r6 * r6);

codelet3(Ei,Eij,E)
  for (jj=0; jj<STEP; jj++)
14| Eij[jj]= Eij[jj] + E[jj];
15| Ei[ii]= Ei[ii] + E[jj];
```

(b) Three codelets obtained after transformation, represented as functions.

Fig. 1. Initial Gibbs hot-spot function and three codelets extracted from it after transformations.

The three codelets exhibit now the following property:

- No dynamic control: Examining the assembly code of the second loop (with the conditional) shows that the conditional has been transformed into predicated code. This implies that now for all three loops, the control no longer dependent on input data (loop branches are statically predicted by icc).
- All data is in cache L2: choosing for `STEP` a value of 100 ensures that data fits in cache, even with the additional arrays resulting from the fission (array expansion on scalars).

– Performance evaluation of codelets do not depend on input values, due to the two previous items.

Now executing the codelets requires to initialize all their input data and depending on the initialization of arrays, in particular of the indirection array `S'`, performance may vary: this may trigger cache conflicts or bank conflicts depending on initial position of new arrays and depending on the values given to `S'`. These effects are not taken into account directly in our model but we measure performance of codelets for different values of `STEP`, preventing cache conflicts cache bank conflicts between simply indexed arrays. We assume these effects can be neglected for the indirection.

Performance evaluation of the three codelets is fast (they contain only one loop, fixed loop bound, data in cache) and the evaluation will help us to define performance evaluation of the whole code, using these codelets. Assume for instance that the three codelets take respectively c_1, c_2 and c_3 cycles to execute for a value 100 of `STEP`. Execution time of the whole code is therefore (in cycles), according to a simple additive performance model:

$$(\texttt{molend} - \texttt{molstart}) * ((\frac{\texttt{molend} - \texttt{molstart} + 1}{100}) * (c_1 + c_2 + c_3) + \epsilon)$$

where ϵ is the time taken by the tail code. It is possible to ignore the effect of ϵ by tiling again the tail code with a tile size smaller that `STEP`. Actual performance measures of the code composed of codelets show in Section 5 that the previous formula predicts performance with 2% accuracy.

Section 2 defines more precisely the characteristics of the codelets we look for, Section 3 describes the framework for the search of these codelets and Section 4 presents the performance model.

1.2 X-language Framework

In order to generate multiple versions of a code, we resort to a two-stage compilation framework. Source-to-source transformations are expressed in a meta-compilation language and are applied on the code after the first compilation stage. The second stage corresponds to a usual compilation phase. We used X-language[10], a language of pragmas, to describe these sequences of transformations. X-language pragmas enable to:

– Specify code fragments (scope) on which X-language transformations apply, using `#pragma xlang begin` and `#pragma xlang end` directives around selected code;
– Trigger source-to-source transformations on code fragment using pragma directives, such as

```
#pragma xlang transform tile(i,II,STRIDE)
#pragma xlang transform unroll(II,UNROLL)
```

The first directive tiles the loop `i` with a stride `STRIDE` into a new loop `II`. The second one partially unrolls `II` by a factor of `UNROLL`. Available transformations include unrolling, tiling, fission, fusion, interchange, scalar promote, ... The rule-based transformation engine of X-language enables more

transformations. Many of these transformations require input parameters, such as the degree of unrolling, the tile size,... Specifying parameter values, and hence generating multiple versions, can be done through other pragmas (extension presented in [3]):

```
#pragma xlang parameter UNROLL [0:8:2]
```

The previous pragma defines values for the unrolling factor `UNROLL` of 0 (no unroll) to 8, with a stride of 2.

– Define new transformations: the rule-based transformation engine of X-language enables simple addition of new transformations or new strategies

The main advantage of X-language is that the user can apply very precisely desired source-to-source transformations.

Extensions to X-language necessary to handle new transformations are presented in Section 3.

2 Constant Performance Codelets

The principle of the method is to explore many code optimizations and by executing the codelets composing a code version, predict performance of this version. We describe in this section the characteristics imposed on these codelets so that the performance model remains simple.

We do not try to model code performance expressed as a function of program inputs. It means in particular that we will not try to predict performance by extrapolation. Performance of codelets are measured in some conditions that will be reproduced in the application so that the measure is reliable. Note that we only focus on cycle counts.

A general formulation of performance would depend on program inputs and on initial machine state. We consider in turn several hardware mechanisms that are difficult to model and the conditions on codes so that their effects can be either measured or neglected. These conditions are mostly on assembly code (this may differ with source code):

1. The codelet is a loop (at least one loop). For large enough loops, the codelet execution takes enough cycles so that impact of the instructions before and after the codelet (through out-of-order mechanism for instance) can be neglected. The "large enough" criteria is architecture dependent. For loops with only a few iterations, our performance prediction may be very different from the real evaluation.
2. Control flow does not depend on input data. It means constant loop bounds, no while loops, and conditional depending only on loop counters. It prevents effects due to branch mispredicts. Note that if..then..else constructs in the source code may be translated into straight line block of code with conditional instructions, for architectures having such instructions (Itanium for instance). In this case, as all instructions of both branches are executed, execution time does not depend on the value of the test (as the example of Gibbs in the introduction).

3. Dataflow dependences in the codelet do not depend on inputs. Indeed, the processor may stall due to detected dependences. If these dependences depend on inputs, performance prediction will lose accuracy. Due to hardware design, dependence testing (a read after a write for instance) is achieved by comparing only a few bits of the addresses. This may cause stalls, even if there is no real dependence. The number of stalls changes only if relative position of starting addresses of arrays changes. We assume that all arrays are aligned to the same boundaries.
4. Configuration of inputs in cache is known when the codelet starts. This condition, combined to the previous ones, ensures that the misses and hits of the codelet will be exactly the same between two executions. However, this assumption implies that for each different configuration of data in cache, a different performance measure has to be made, thus limiting the number of configuration. In our examples, we considered only the case when data was fully contained in a cache level or not.
5. No I/O or system calls.

These conditions define a *constant performance codelet* and the search aims to decompose the code into these codelets. As a consequence of their definition, these codelets can be taken out of the application context and benchmarked *in vitro* to evaluate their performance in different input configurations (cache hierarchy that contain these inputs). Compared to traditional iterative compilation framework, there is no need to execute the whole application.

The identification and the extraction of the codelets are nowaday still performed by the programmer. We focused our research on the loops which take the major amount of time in our programs. To automate this part of the work, a simple code extractor should be coupled with a profiler tool to extract only the more interesting loops top optimize, or at least to give some hint to the programmer on which loops the search should be apply. Already existing code isolators tend to collect a print of the machine state when executing the codelet in the whole program to offer the same runtime environment for the isolated code [23]. Our method allows the extracted codelet to be independant from the program environment, and, at the end, it will be the program to be adjusted according to the more performant optimized codelet.

Extracted codelet can be just considered as new macro instructions, specific to the application. There is no parallelism between these instructions (no ILP, no out-of-order effect) and as long as the instructions are executed with their data in the right memory configuration, the performance model is just additive. The total execution time is the sum of the measure latencies of each instruction.

The following section describes how these codelets are generated and Section 4 explains how to make sure that the codelets are always executed in the same memory configuration as they were executed.

3 Searching for High Performance Codelets

The goal of this search is to search among all code transformations those that will lead to the definition of high performance codelets. As seen in previous section, a constant performance codelet has to check some static properties that help define the search. Due to the potential high number of codelets, a static analysis on the compiler-generated assembly code reduces the number of required executions (discarding codelets with obvious poor performance).

3.1 Defining search space with X-language

X-language relies on a C99 compiler `tcc`[1] that passes an AST to a Prolog program that does the effective search, based on a set of rules. X-language has been used to generate many versions of codes for linear algebra libraries[3]. For more complex codes, some extensions have been added, in the pragma language itself and in the range of transformations considered. involved, pattern guided vectorization.

X-language does not have by itself a dependence analysis, this removes a constraint due to overly conservative dependence analysis, but may also generate incorrect code. For some transformations (such as slicing) it is necessary to indicate the group of statements that must be kept in sequence. This is achieved by two pragmas:

- `#pragma xlang section` id where n is a unique number. This indicates the beginning of sequential section of code.
- `#pragma xlang dependence`$(list - of - ids)$. This indicates that a chain of dependence between sections of code.

The following transformations are added to X-language:

- `#pragma xlang transform unrolljam`$(loop - id, unroll - factor)$: unroll and jam is a simple extension based on existing transformations and dependence analysis provided by previous pragmas. The jam performs fusion of all loop surrounded by the unrolled loop.
- `#pragma xlang transform vectorize`$(loop - id)$: vectorization is pattern based and applies only if all statements of a loop can be vectorized. Vectorization is decomposed into as many pattern based rules as there are vector instructions.
- `#pragma xlang transform tryintrinsics`$(name)$: it matches all function names and replaces them with the equivalent intrinsics (if present in rules). This directive will generate codes with and without the intrinsics substitution.
- `#pragma xlang transform distribute`$(loop - id)$: distributes statements in different loops, preserving dependences given by the previous pragmas.

For all applications, two parameters are searched for: tile sizes (generating inner loops with constant loop bounds) and unrolling factors.

Here is an example of the X-language program used to explore different versions of GIBBS and generate multiple codelets.

```
#pragma xlang parameter SIZE [64:256:64]
#pragma xlang parameter UNROLLF[0:8:2]
#pragma xlang parameter UNROLLI[0:4:2]
#pragma xlang parameter any(tile1D(j,jj,SIZE),tile2D(i,j,ii,jj,SIZE,SIZE))
#pragma xlang transform tryIntrinsics(round)
#pragma xlang transform unrollandjam(ii,UNROLLI)
#pragma xlang transform distribute(jj)
#pragma xlang transform unroll(jj,UNROLLJ)
#pragma xlang transform decompose(1)
```

with the dependences accordingly. The keyword "any" defines that any combination of transformation given as parameter will be tried. A number of rules define optimization in the search, and in particular combining tilings is not permitted. Therefore here the directive just states that one or the other tiling should be tried. The last directive selects inner loops that fulfill conditions for constant performance codelets and exports them as C files, ready to be compiled separately.

Some phases of the search, notably the writing of initializing code for the codelet, are still performed by hand.

3.2 Evaluation of codelets

Before benchmarking the codelets, we must checked that the performance measured will be stable in the application context. In particular, the compiler generated assembly file is checked with a tool we developed, MAQAO [9].

For codelets that still have some conditional depending on variables other than loop counters, MAQAO checks whether the assembly loop body has one basic block or not. In which case, the conditional has been replaced by predicated code. Otherwise, the codelet is discarded. Some codelets exhibit poor code quality. For instance, speculative codes for Itanium usually offer poor performance: the compiler speculates that there is no dependence between several statements but is unable to prove it. Spill code due to high register pressure is also detected. In these cases, the code is discarded before execution.

Since the codelets are rather small portion of code, they are executed several times, with the data already in the right memory configuration before the measurement, to avoid timers problems [35].

Once the codelets have been tested, possibly in different input cache configuration, the final code has to be built, ensuring that the codelets are executed in the same condition. Two additional codelets are generated automatically: copies and prefetches, as described in the following section.

4 Memory Hierarchy Optimization

The goal is to transform the code using the codelets so as to ensure that codelet performance will be the same than the performance measured. As codelets assume their data is in some cache, this requires some cache model in order to

compute the data region necessary to prefetch or copy before execution of the codelets. However, the objective is not to modify the codelets by adding prefetch instructions inside, as this would alter performance. On the contrary, prefetches and copies will be added by blocks. Performance of these blocks can also be measured.

We use a simple memory model based on cache capacity. If the data structure of the codelets is altered (due to vectorization for instance), a block of copies is chosen otherwise a block of prefetches is taken. These codelets are first schedule right before the codelet and then moved at earlier positions as long as the working set between the copy and the codelet still fit in cache. This model is very preliminary but we found that it was sufficient to handle the application considered. The reason is that all applications considered access data with indirection. In this case, precise evaluation at compile time of data movement is no longer possible.

We need to measure block of copies and of prefetches designed for particular codelets. Let us whether they can be considered as constant performance codelets. They have constant loop bounds, static control, and a loop. Moreover, dataflow dependences inside the codelets do not depend on inputs. Different tests are needed, according to the position of the data to copy/prefetch in the memory hierarchy. However, performance still depends on spacial locality since two data in the same cache line and accessed consecutively will be accessed faster than if they were mapped to two different cache lines. Variation between two executions with different input data (not size) only happen for A[B[i]], due to this effect. When an indirection of the kind A[B[i]] occur in a copy, we choose the worst case execution and initialize B such that it addresses different cache lines.

For blocks of prefetches, we design a codelet that prefetches all data and add to it a small kernel consuming this data as fast as possible (with a reduction). When the data prefetched is not in cache, an evaluation of this codelet provides an evaluation of the latency due to the miss, and the overhead of the reduction itself. It is important to notice that such case can be considered as a worst case, since data prefetched is accessed right after the prefetch. In an application a more favorable case may occur. Figure 2.b shows the performance of a codelet of prefetch for 15.10^9 elements (spinors) required by a codelet from the HMC application. The left bar measures only the overhead of the codelet, when data is already in cache. The right bar shows the latency of the memory access, when data in memory is prefetched and then used. This latency minus the overhead provides the performance measure (worst case) for prefetches.

5 Experimental Results

We present application of our method to three applications.

5.1 Experimental setup and Applications

The target architecture is a BULL Novascale server featuring 1.6GHz Itanium 2 processor, with 3 cache levels was used. Out of these 3 levels, only the 2nd level

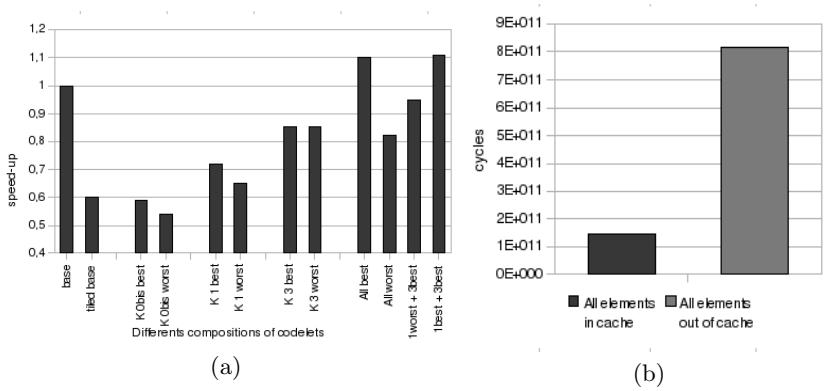


Fig. 2. (a) Performance estimations for Gibbs depending on the combination of codelets involved. (b) Performance measures for a codelet prefetching a spinor structure for HMC. The codelet makes a block of prefetch followed by a reduction consuming prefetched data. The measure is taken when all data is in cache (measuring overhead of codelet) and when all data is in memory.

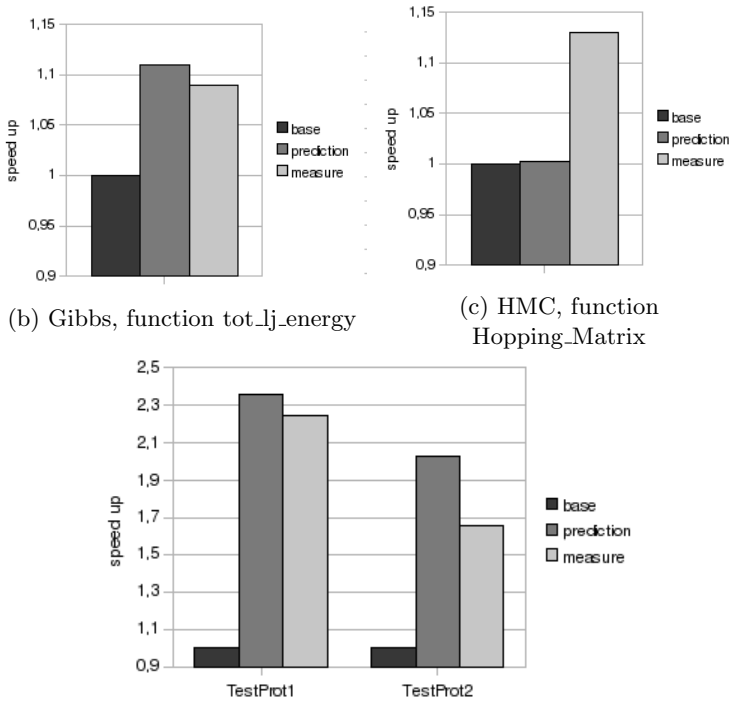
(256KB, unified) and the 3rd level (9MB, unified) can contain floating point values. The processor offers 128 floating point registers and can issue up to 6 instructions per cycle.

We applied our technique on three applications. GIBBS, described in the introduction, is a molecular simulation. The inputs considered required several hours of computation. BLAST is genomic application for the comparison of sequences of genes. The implementation we used is described in [22] before being adapted to an FPGA accelerator. HMC is a simulation code for Lattice Quantum Chromodynamic (LQCD). LQCD is a numerical method based on Monte Carlo method for the study of the theory of strong interaction in the domain of subnuclear physics. Simulation of scientific interest require several thousand hours of computation on parallel supercomputer. The input set we used for our experiment is realistic enough for a sequential code and takes 5 hours on one core of Itanium2. The HMC implementation we used is achieved by the ETMC collaboration[32]. For each of these application, we focus only on the hot spot function (there is only one for all these cases).

5.2 Comparing Estimated Performance and Measured Performance

Searching efficient codelets first needs to define a pragma program in X-language driving the search. For each application, we list the optimization that were necessary to obtain the presented speed-up. For Gibbs: loop distribution, tiling, unrolling and substitution of a mathematical function by intrinsics. For BLAST: vectorization (using intrinsics), unroll and jam, unrolling, memory copies. For HMC: loop distribution, tiling, prefeches, unrolling.

Figure 3 shows the performance speed-up predicted by our performance model, when using the best codelets, for the main computation function of the applications. This is to be compared to the real measured performance.



(b) Gibbs, function tot_lj_energy

(c) HMC, function Hopping_Matrix

(a) BLAST, function BLAST

Fig. 3. Performance of initial code (base 1, labeled origin), of the optimized code according to performance prediction (speed-up labeled preview) and of the optimized code according to performance measure (speed-up labeled measure).

For BLAST, vectorization and some unrolling brings most of the performance gain. There is a gap between predicted performance and measures for Test2 because it is assumed that data is in cache for prediction while this is not exactly the case for this input. For Gibbs, there is a 10% of speed-up according to performance prediction and 9% of observed gain. For HMC, according to performance prediction, there is no speed-up when using the best codelets whereas there is a 10% speed-up in the real code. The reason is that there is some reuse of data from one call of the function optimized, Hopping_Matrix, to the other and this is not taken into account in the performance prediction.

6 Related Works

High performance libraries is the key for high performance applications. ATLAS [36] for instance focuses on linear algebra computations. But even with different improvements [39], vendor[11, 25], hand-tuned BLAS3[15] still outperform ATLAS compiled codes and, up to now, such libraries manage to reach near-peak performances on linear algebra kernels. Library generators resort to many optimizations that usually are not part of the optimization sequence of a compiler. Among them, loop tiling [37, 38] and register blocking [17, 4] improve data locality on all levels of the memory hierarchy. The objective of these optimizations is to decrease cache misses.

In our approach, we choose instead to make a tradeoff between Instruction Level Parallelism (ILP) and cache usage, with a method of kernel decomposition [3] which divides the program into code fragments. These code fragments called codelets are used to obtain high performance. Unlike telescoping languages [5, 18], in our approach, the different language levels collaborate: source transformations explore high level cache optimizations that are difficult for production compilers, and the approach relies on the compiler to express the ILP.

Adaptive and Iterative Compilation: In order to obtain good performance when running on a specific machine, several adaptive [34, 33] or iterative [19, 29] compilation techniques has been implemented. The principal drawback of these techniques is the necessity to run the whole program several times to obtain the best performance. Even when different optimized codes can be tested in the same run [14], the whole application has to run at least once to detect the best version. This is not realistic for very long-running program. Our method allows to do some iterative compilation on a smaller part of the program, which is less time consuming.

Extraction and Running Context: Despite the fact that code extractors and isolators already exist in the litterature, we developped our own method to extract codelets for optimization. The main reason is that they tend to not only extract the code, but also save the state of the hardware when the original to-be-extracted code is running in the whole program [23]. Since our method is based on a simple memory model, we only need to know how the caches are filled with the codelet data.

Annotation and Pragma Meta-language: To express the different optimizations to apply on the codelets, we choose a meta-language based on pragmas to automatically generate the different versions of the codelets. Since other works allowed to make the same type of transformations [27], our choice being this language was practical: it has been mainly developped in our team [10] and, as this part is not the main contribution of this paper, we used the language we had at hand.

The main drawback of our method being search space can become potentially huge, it should be interesting to couple it with some techniques to shrink it. For example, we could search a smaller space using genetic algorithms and machine learning techniques [7, 8, 26], or scan more points withing the same amount of time. With these methods, the iterative search for codelets would be decreased.

Memory Model: Our memory model, though very simple, was sufficient for us to realize accurate predictions. Other detailed models exist ([12] for instance), describing cache behaviour of individual memory access. But as codelets work on chunks of arrays that are large enough, an approximated cache model is enough for us. Some Models simply don't address the same problem as us [24] using their model to select the best variant of codelet, when we use our model to predict precisely the execution time of the optimized function. In fact, this kind of model could be used to filter out statically codelets with bad memory performance prediction.

7 Conclusions

The approach presented in this paper proposes to address long compilation times of adaptive compilation. The goal of the search we propose is to decompose the code into high performance codelets that can be tested outside of the application context. We have shown that on three real applications, from the performance evaluation of these codelets, a simple performance model is able to predict with accuracy the performance of the optimized code. While there is still many required execution of codelets, there is no need for long execution of the whole application.

Our method can be improved in a number of ways. We have assumed that the user is able to guide the optimization search, defining an appropriate optimization parameter search space for the application. A search base on the characteristics of the code would be a significant step towards the integration of this technique into an optimizing compiler. A more complex and realistic cache model would increase the range of codes for which the performance model is accurate . In particular, modeling n-way associativity, TLB misses, considering potential conflict misses between array sections would be interesting in order to apply our technique to more general codes, such as the hot spots of applications.

References

1. Tiny C compiler. http://www.tinycc.org.
2. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. Oboyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *In Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305. IEEE Computer Society, 2006.
3. D. Barthou, S. Donadio, A. Duchateau, P. Carribault, and W. Jalby. Loop optimization using adaptive compilation and kernel decomposition. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 170–184, San Jose, California, Mar. 2007. IEEE Computer Society.
4. C.-Y. Chang, J.-P. Sheu, and H.-C. Chen. Reducing cache conflicts by multi-level cache partitioning and array elements mapping. *J. Supercomput.*, 22(2):197–219, 2002.
5. A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *ACM/IEEE Conf. on Supercomputing*, pages 92–101, June 2001.

6. K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *In Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9. ACM Press, 1999.
7. K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.
8. K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 23(1):7–22, 2002.
9. L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. A new tool for assembler analysis and optimization on epic architecture. In *Proc. of the Epic Workshop (in conjunction with CGO'05)*, 2005.
10. S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of *LNCS*, Hawthorne, New York, Oct. 2005.
11. Engineering and scientific subroutine library. Guide and Reference. IBM.
12. B. B. Fraguela, R. Doallo, J. Tourino, and E. L. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Computing*, 30(2):225 – 248, 2004.
13. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the ICASSP Conf.*, volume 3, pages 1381–1384, 1998.
14. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *High Performance Enmebdded Architectures and Compilers*, pages 29–46. Springer Berlin / Heidelberg, 2005.
15. K. Goto and R. van de Geijn. On reducing tlb misses in matrix multiplication. Technical report, University of Texas, 2002.
16. IBM. Exploiting the dual floating point units in blue gene/l. IBM developper support.
17. M. Jimenez, J. Llaberia, and A. Fernandez. Register tiling in nonrectangular iteration spaces. *ACM Trans. Prog. Lang. Syst.*, 24(4):409–453, 2002.
18. K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proc. Intl. Parallel and Distributed Processing Symp.*, pages 297–304, 2000.
19. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.
20. T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2000. IEEE Computer Society.
21. P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI'04, proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 171–182, 2004.
22. D. Lavenier, X. Xinchun, and G. Georges. Seed-based genomic sequence comparison using a fpga/flash accelerator. In *Intl. IEEE Conference on Field Programmable Technology (FPT)*, 2006.
23. Y.-J. Lee and M. Hall. A code isolator: Isolating code fragments from large programs. pages 164–178, 2005.
24. N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In *In V.N.Alexandrov, J.J. Dongarra, Computer Science*, pages 28–30. Springer, 2000.

25. Intel math kernel library (intel mkl). Intel.
26. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.
27. B. Norris, A. Hartano, and W. Gropp. Annotations for productivity and performance portability. In *In Petascale Computing: Algorithms and Apllications*, pages 443–462. CRC Press, 2007.
28. N.Thomas, G.Tanase, O.Tkachyshyn, J.Perdue, N.Amato, and L.Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proc. ACM PPOPP'05*, Chicago, 2005.
29. M. O'Boyle, P. Knijnenburg, and G. Fursin. Feedback assisted iterative compiplation. In *Proc. LCR*, 2000.
30. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.
31. R. Singh, K. Pitzer, J. D. Pablo, and J. Prausnitz. Monte carlo simulation of phase equilibria for the two-dimensional lennard-jones fluid in the gibbs ensemble. In *The Journal of chemical physics*, volume 92. American Institute of Physics, 1990.
32. C. Urbach. Lattice QCD with Two Light Wilson Quarks and Maximal Twist. *The XXV International Symposium on Lattice Field Theory*, 2007.
33. M. J. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 93–102, New York, NY, USA, 2001. ACM.
34. M. J. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *In Proc. ICPP*, pages 163–170, 2000.
35. R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38(15):1621–1642, 2008.
36. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.
37. M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Aug. 1992. Published as CSL-TR-92-538.
38. M. Wolfe. Iteration space tiling for memory hierarchies. In *Conf. on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, 1989. SIAM.
39. K. Yotov, X. Li, G. Ren, M. Garzarán, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLASs? *Proc. of the IEEE*, 93(2):358–386, February 2005. Special issue on "Program Generation, Optimization, and Adaptation".