HAL
open science

Compositional Approach applied to Loop Specialization

Lamia Djoudi, Jean-Thomas Acquaviva, Denis Barthou

To cite this version:

Lamia Djoudi, Jean-Thomas Acquaviva, Denis Barthou. Compositional Approach applied to Loop Specialization. Concurrency and Computation: Practice and Experience, 2009, 21 (1), p71-84. 10.1002/cpe.1337 . hal-00575934

HAL Id: hal-00575934

https://hal.science/hal-00575934v1

Submitted on 28 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compositional Approach applied to Loop Specialization

L.Djoudi, J.-T. Acquaviva, and D. Barthou

Université de Versailles, France
{lamia.djoudi,denis.barthou,jean-thomas.acquaviva}@uvsq.fr

Abstract. An optimizing compiler has a hard time to generate a code which will perform at top speed for an arbitrary data set size. In general, the low level optimization process must take into account parameters such as loop trip count for generating efficient code. The code can be specialized depending upon data set size ranges, at the expense of code expansion and decision tree overhead.
We propose for loop structures a new method to specialize code at the assembly level, cutting drastically the overhead cost with a new folding approach. Our technique can generate and combine sequentially at the assembly level several versions, tuned for small, medium and large iteration number. We first show on the SPEC benchmarks the need for specialization on small loops. Then we demonstrate the benefit of our method on kernels with detailed results.

1 Introduction

An optimizing compiler has a hard time to generate a code which will perform at top speed for an arbitrary data set size. In general, Schwiegelshohn *et al.*[1] have shown there is no one best scheduling function for a loop, for all possible data sets. Even for regular programs, the best latency is only reached asymptotically [2,3], for large iteration counts. Splitting loop index to obtain better schedules[4], or tiling iteration domains are well known techniques that improve latency. These transformations are driven according to source code features such as dependencies or memory reuse. On the other hand, low level optimizations must take into account parameters such as loop trip count for generating efficient code: for example, short loop trip count would favor full unrolling while very large loop trip counts will favor deep software pipelining. To some extent, the code generated has to be specialized depending upon data set size ranges and then has to use extensive versioning to apply these different specialized versions. The classical drawback of such an optimization scheme is code expansion and decision tree overhead. It usually puts a hard limit on the total number of different specialized versions generated.

We propose, for loop structures, a new method to specialize code at the assembly level and to drastically cut the overhead cost with a new folding approach. Taking the assembly code, we are able for instance to generate three versions tuned for small, medium and large iteration number. We combine all

these versions into a code that switches smoothly from one to the other while the iteration count increases. Hence, the resulting code achieves the same level of performance as each version on its specific iteration interval.

We first show on the SPEC benchmarks the need for specialization on small loops. Then we demonstrate the benefit of our method on kernels optimized with software pipeline, with experimental results.

1.1 Motivating Example

Loop optimization is a critical part in the compiler optimization chain. A routinely stated rule is that 90% time of the execution time is spent in 10% part of the code. Another rule, implicitely used by the community, is that the number of iterations for loops in scientific code is *large*. Consequently, loops are often unrolled, pipelined deeply and data streams aggressively prefetched.

However, optimizations for asymptotic behavior involve a part of risk. For instance in software pipeline, depth is always increased if it can reduce the Initiation Interval. This yields to codes which deliver poor performance when the number of iterations is limited. Figure 1 clearly illustrates the trade-off that the compiler has to handle on a simple vector loop named Tridia. ICC 8.1, first unrolls this loops two times and generates a software pipeline of depth 2. While ICC 9.0 unrolls this loops 8 times, then applies software pipeline. The corresponding tail code is also software pipelined.

The corresponding performance evaluation is:

- ICC 9.0: $65 \times \frac{N}{8} + 130$ (unrolled 8 times) and $10 \times (N \bmod 8) + 20$ for tail code.
- ICC 8.1: $24 \times \frac{N}{2} + 48$ (unrolled 2 times) and $14 \times (N \bmod 2)$ for tail code.

As illustrated by figure 1, ICC 9.0 choice is justified for asymptotic performance but is doubtful when the number of iterations is small.

To evaluate the importance of the short loops we have performed a set of measurements on the SPEC FP 2000 benchmarks. Using MAQAO tool [5], all loops are instrumented. Instrumentation is done at the assembly level to prevent distortions in the compiler optimization chain. This instrumentation simply measures the number of iterations executed per loop and the number of CPU cycles spent within the loop. At the end, histograms are built according to the number of iterations of these loops weighted by their execution time. We use ICC v9.0 with flags reported for SPEC results, including profiling guided optimization, on a 1.6 GHz / 9 MB L3 Itanium 2 system. The only instrumented loops are counted loops, software pipelined loops, but loops driven by conditional branches are currently not caught by our tool.

Additionally the numbers provided should be considered knowing that ICC performs aggressive unrolling (most of the time by a factor of 8), consequently reducing the number of loops iterations.

Figure 2(a) details measurements made on the number of iterations for loops of CFP2000 codes using the `ref` data set. The answer is surprising: 25% of loop

```
DO 1 I=2,N
  CP1=1./(CP(I)-CI(I)*CS(I-1))
  CS(I)=CS(I)*CP1
  CSM(I)=(CSM(I)-CI(I)*CSM(I-1
1 CONTINUE
```

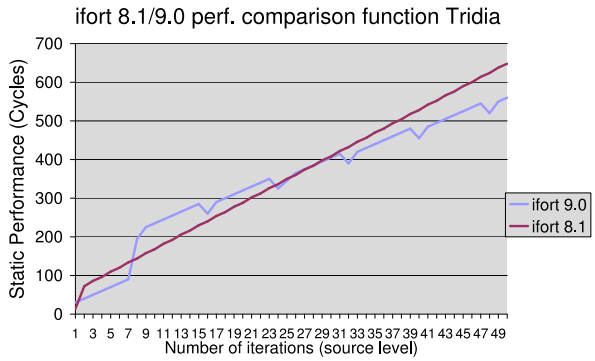
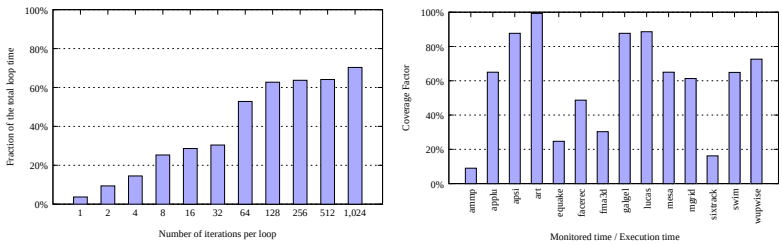


Fig. 1. Tridia code and its performance with ICC version 8.1 and 9.0. Moving from version 8.1 to 9.0, ICC has changed part of its code generation policy. None of the two different versions is optimal over the whole possible range of iterations.

time is spent in loops with less than 8 iterations. A more detailed analysis shows that 6 over 14 benchmarks form the CFP2000 spend half of their loop time in loops with less than 16 iterations. Therefore, loop tuning based on infinite number of iteration is missing real performance opportunities, and compilers should not over-simplify loop behavior. In order to back the idea that short loops



(a) SPEC FP overall cumulative loop distribution

(b) SPEC FP overall cumulative loop distribution

Fig. 2. (a) Percentage of execution time spent in loops with iteration count $\leq$ x-label. (b) Fraction of the total execution time spent in loops for each benchmarks of the SPEC FP 2000 suite. Coverage factor is computed as the number of cycle spent in instrumented loops over the total number of execution cycles.

are an important problem, Figure 2(b) details the fraction of the execution time spent in loops benchmarks in SPEC.

2 Compositional Approach

The idea of the approach, given different codes (and schedules) for the same loop, to combine or compose them into one code that achieves the same performance as the best code, for any iteration count. The purpose of this method is to assemble together in a costless and smart way different versions of the same loops. It is understated that the quality of the resulting composed version will depends on the quality of the different individual versions available at the origin.

Iterative compilation is an approach that relies on the generation of many different versions of the same code to find out the best among them. Loops generated by iterative compilation are therefore good candidates.

2.1 Iterative Compilation Framework

Iterative compilation is decomposed usually into two steps: (i) Generate multiple optimized versions of the same code. The goal is to generate a small number of versions that have good performance while covering a wide range of iterations. (ii) Compare the performance of these different versions (either by a model or by a dynamic evaluation) and build a program combining them. Most of the research effort has been on the first step, and for the second one, it usually boils down to generate a decision tree. This means that for one execution, only one of the specialized code is executed. In this paper, we focus on this second step and generalize the previous case by enabling one execution to execute several specialized codes in sequence. Indeed, optimization may be beneficial only on a given range of iterations. For each of the following optimizations, we describe its limitations and the conditions for which it applies

1. *peeling*; Peeling enables a rescheduling of the first iterations of a loop. It generates more opportunities for a better resource usage, with a free schedule, at the expense of code expansion.
2. *unrolling*; Unrolling a loop body offers the opportunity for better ILP. The higher the unrolling factor, the higher the impact on performance of the tail code for small loops.
3. *data prefetching*; Data prefetching cuts by a large amount the read/write latency of memory accesses. Tuning the prefetch distance is highly dependent on the total number of iterations. For small loops, the prefetch distance is too large for the prefetching to be effective. In this case, removing the prefetches may free resources for a better ILP, therefore increasing performance.
4. *software pipeline*; The Initiation Interval (II) is usually the value minimized by software pipeline algorithms, and represents the amount of time between two successive start of iterations. This comes at the expense of the latency required to execute a complete iteration, which is important for small loops.

This shows that there are many opportunities in which it would be interesting to combine different optimized codes according to the iteration count.

2.2 Performance Model

Consider two different optimized versions of the same loop, called L_1 and L_2. This can be generalized to any number of versions. We assume that these loops are inner loops (they do not include other loops). The cycle count of L_1 is given by the formula: $\mathrm{c}_1(i) = \alpha_1.i + \beta_1$, where α_1 is an rational number, β_1 an integers and the cycle count is rounded down. Similarly for L_2, $\mathrm{c}_2(i) = \alpha_2.i + \beta_2$. For instance, for Figure 1, the cycle count for the loop L_1 generated by ICC 8.1 is defined with $\alpha_1 = 12, \beta_1 = 48$ and for the loop L_2 generated by ICC 9.0, $\alpha_2 = 8.125$, and $\beta_2 = 130$, without the tail code. Tail code is considered for our purposes as another version of the code, within the range of 7 or 1 iteration (for the loops unrolled 8 and 2 times respectively).

We consider the case where the two loops are such that: $\alpha_2 < \alpha_1$ and $\beta_1 < \beta_2$, meaning that L_1 is faster than L_2 when $i < \frac{\beta 2 - \beta 1}{\alpha_1 - \alpha_2}$ and L_2 outperforms L_1 for larger number of iterations. We would like to build a *best* code such that:

$$\forall n, \mathrm{c}_{best}(n) = \min_k(\mathrm{c}_k(n)).$$

This *best* code is built by an optimization function min: $\min(L_1, L_2) = best$. This function min defines a minimum on codes with respect to the performance, for all iteration values. Due to the difficulty of building the minimum of two codes without introducing any overhead, we propose to tackle a more pragmatic problem. We want to build a code $\min(L_1, L_2)$ with a level of performance very close to the performance of the best of the two codes. The following constraints are applied to the loop to build:

1. Asymptotic performance (in cycle/iteration, when iteration count grows) is the same than the best asymptotic performance of L_1 and L_2.
2. Each loop is possibly called many times, each time with a possible different loop trip count. Given a loop count distribution, the average gain in cycle/iteration compared to the best asymptotic performance of L_1 and L_2 is positive.
3. When performance of both loops L_1 and L_2 meet, the loop built moves to the best code for asymptotic performance.

Note that the second constraint does not compel the new loop to outperform L_2 and L_1 for each iteration count, but in general, for all the execution of the loop, some cycles have been gained. The reason is that some overhead may appear when switching from one version to the other. The best code would have the following cycle count:

$$\mathrm{c}_{12}(i) = \begin{cases} i \leq B : \mathrm{c}_1(i) \\ i > B : \mathrm{c}_2(i) - \mathrm{c}_2(B) + \mathrm{c}_1(B) + \gamma \end{cases}$$

where B is the integer $\frac{\beta 2 - \beta 1}{\alpha_1 - \alpha_2}$ and γ represents the overhead necessary when going from one version to the other. This overhead represents register initializations, branch mispredicts,.... The difference in cycles/iteration between the

asymptotic best loop and the new loop $\min(L_1, L_2)$ is, for an iteration count i:

$$dpi(i) = \begin{cases} i \leq B : (\mathrm{c}_2(i) - \mathrm{c}_1(i))/i \\ i > B : (\mathrm{c}_2(B) - \mathrm{c}_1(B) - \gamma)/i \end{cases}$$

The difference in cycle/iteration is asymptotically 0, meaning that this new version is as fast as L_2. When the loop iteration count is uniformly distributed among iterations $[1..N]$, the average difference in cycle/iteration is obtained by:

$$adpi(N) = \sum_{i=1}^{N} \frac{dpi(i)}{N}.$$

This definition can easily be adapted to other distributions. In particular, distribution of values caught during profiled execution can be used. When $adpi(N)$ is positive, it means that for a uniform distribution of loop trip counts in $[1, N]$, the new loop $\min(L_1, L_2)$ is in average faster than the best asymptotic loop L_2. This value is positive when $N < B$ since each difference/iteration is positive for all iterations $i < B$. For higher values of N, $adpi(N) > 0$ if:

$$\gamma < \frac{B(\alpha_2 - \alpha_1)(1 + H(N) - H(B)) + (\beta_2 - \beta_1)H(N)}{H(N) - H(B)}, \tag{1}$$

where $H(N)$ is the harmonic number $H(N) = \sum_{k=1}^{N} \frac{1}{k}$. As H is a strictly increasing function, when N asymptotically grows, γ must be such that:

$$\gamma < \mathrm{c}_2(B) - \mathrm{c}_1(B).$$

This constraint implies that the new loop $\min(L_1, L_2)$ takes less cycles than the best asymptotic version, for any value of the iteration count. From this constraint we can deduce the basic steps to build the code $\min(L_1, L_2)$:

1. Compare $\mathrm{c}(L_1)$ and $\mathrm{c}(L_2)$, in order to compute B
2. Assuming L_1 outperforms L_2 for the first B iterations, evaluate the code in-between necessary for the transition and the overhead β generated.
3. If inequality 1 is satisfied, then build the minimum of the two codes. Otherwise the overhead is too significant w.r.t. the total execution time.

For the example in Figure 1, $B = 21$, inequality (1) entails that for $N > 293$, γ can no longer be strictly positive. For $N = 200$, γ can be up to 15 cycles.

2.3 Scopes and Limits

As with all other versioning schemes, our approach improves performance at the expense of code size. If the number of iteration remains constant or at least in a single range the extra code size and some instruction overhead will penalize the execution time. However if we considere the SPEC benchmark as representive of the average code complexity it be can safely stated that iteration range is varying a lot and that specialization on iteration number will mostly inscreases performance.

3 Assembly to Assembly Transformation

We first present an assembly code dependence analysis then describe two particular transformations, loop peeling and transformation of prefetching, as well as their composition. These two steps are for an Itanium architecture, but we believe this can be generalized to other platforms. Such post-compiler optimization is already a hot topic of research [6,7,8].

3.1 Code Flattening and Dependence Graph

In order to preserve code semantics, the validity of the transformations applied is checked by computing a *data dependence graph* (DDG) on the assembly codes. Dependencies considered can be either intra-iteration or inter-iteration and dependence analysis is required by the peeling and jam transformations described in the following section.

In the case of pipelined loops on Itanium, dependence analysis is more complex and loop flattening is a preliminary transformation. IA64 Hardware support for software pipelining includes a rotating register bank and predicate registers [9]. Loop flattening is a transformation that removes the effect of software pipeline: it renames carefully registers according to their rotating status or not, and predicates are used to retrieve the iteration number when the instruction becomes valid.

A data dependence graph is then built between the different instructions in the loop. Register dependences are built by reaching definition analysis. For memory accesses, the alias analysis performed relies on two techniques: We apply a conservative approach, based on the schedule generated by the compiler. The base rule is that all memory accesses are interdependent (read or write with write). If the compiler schedules two instructions within less cycles than the minimum latency of the first instruction (the one being the possible dependency anchor) then we assume that the compiler did this schedule on purpose and therefore that the two instructions are independent. For instance, if a load `ld f32,[r31]` is scheduled 3 cycles before a store `st [r33],f40` and the minimum latency of a load is 6 cycles, then both statements are independent.

We also resort to a partial symbolic computation of addresses, using induction variable analysis on address registers. The value of address registers can often be computed with respect to some initial parametric values coming from registers defined outside of the loop (parameters of a function for instance). In this case, our de-ambiguation policy depends on the original compilation flags (either with or without no-alias flag). More independent statements can be found that way.

3.2 Peeling and Prefetching Transformations

We show the composition approach using loop peeling and prefetching.

Peeling is the process of 'taking off' a number of iterations from a loop body, and consequently explicitly express them at the beginning or end of the loop. This is often done in order to match two different bounds of two subsequent

loops. Generally, the positive effect of this technique is better understood if explained in conjunction with loop fusion. In our approach, the peeling has also a positive effect if explained in conjunction with software pipelining. Compared to warming up stages of software pipelined loop, an interleaving scheme does not increase latency but increases the number of iterations simultaneously in-flight. This does not yield to excessive register pressure. In fact, the global register pressure depends on the number of iterations simultaneously alive. Our peeling techniques is careful enough to keep this number below the software pipelined loop asymptotic behavior. The initial schedule of peeled iterations is the schedule obtained after a possible flattening. Then iterations are jammed (or interleaved) with a list scheduling algorithm with priority to the first iterations. The statements of the first iteration peeled are scheduled first and have higher priority over the statements of the second iteration. Indeed, this schedule improves over the initial schedule, w.r.t. the difference in cycle per iteration, as presented in Section 2.2. If the initial (flatten) loop has a cycle count function of the form $\alpha.i+\beta$, a jammed version of the peeled iterations takes $c_{peeled}(i) = \alpha'.i+\beta$ cycles with $\alpha' \leq \alpha$ where α' is a rational number. The list scheduling algorithm ensures that the longest dependence chain in one iteration is not increased, the latency α' is lower or equal to α. Finally, a mechanism is needed to ensure program correctness if the number of total iterations is smaller than the number of peeled iterations. One calculated branch is used for a late entry into the peeled code, and predicate registers guard interleaved instructions. The branch uses a branch register to store the address to jump in. Setting a value to a branch register is a 6 cycle long operation. If the execution time of interleaved iterations exceeds 6 cycles we use this kind of register to minimize the overall latency. Moreover, instructions guarded by predicates prevent from executing interleaved instructions that do not belong to the desired peeled iterations. $\log_2(N)$ comparisons, and $\log_2(N)/6$ cycles are necessary to set the predicate registers of N peeled iterations. The overhead is limited to a couple of cycles.

For prefetching, the prefetch distance is estimated from the symbolic computation performed before and from the increment of the address registers, we assess the number n of first loop iterations that do not take advantage of the prefetch. The loop is then split into a sequence of two similar loops. The first loop has no prefetches (they are replaced by nops) and has n iterations. The second loop is the initial loop, performing the remaining iterations.

4 Related Work

Specialization is a well known technique to obtain high performance programs. Compiled time specialization often boils down to the generation of codes that are in mutually exclusive execution paths. Splitting iteration space in order to apply different optimizations for each fragment has been proposed by Griebl *et al.*[4]: their goal is to partition the iteration space according to the dependence pattern for each statement. This increases control but increases the number of affine schedules that can be computed for each code. Tiling is another transformation

that changes the iteration domain for better scheduling. However, very few works resort to loop versioning in order to explicitly reduce the overall latency of the loop. This is due to the intractability of general performance models (finding best latency affine schedules is still a difficult problem). That is one reason why asymptotic loop counts are generally considered for optimization. In this paper, we do not consider memory models (or cache model) and assume that the compiler or a tool evaluates accurately the performance of inner loops. This is easier on assembly than on source code.

Software pipelining[10] is a key optimization for VLIW and EPIC architectures. In particular, modulo scheduling, as used by the `ICC` compiler, exhibits instruction parallelism, even in the presence of data dependencies, that greatly improves performance. Modulo scheduling targets large iteration counts and tries to find an initiation interval (II) as small as possible, defining the throughput of the loop. However, when the iteration count is small, this may increase the loop latency (in particular when the II is essentially constrained by resources). Loop peeling is a well known technique for improving the behavior of the code for small iteration count. As it comes at the expense of code size, compiler heuristics usually prefer to not use it. With our approach, it is possible to decide, according to the awaited iteration count distribution, whether peeling is worth or not. Moreover, our technique would take advantage of profile information since the distribution is then more accurate.

Prefetch works by bringing in data from memory well before it is needed by a memory operation. This hides the cache miss latencies for data accesses and thus improves performance. Typically, when a cache line is brought in from memory, it contains several elements of an array that is accessed in the loop. This means that a new cache line is needed for this array access only once in several iterations. This depends on several factors such as the stride of the array access pattern, the cache line size, etc. In fact, if redundant prefetch instructions are issued every iteration, it may degrade performance. Prefetch instructions require extra issue slots and thereby increase the cycle per iteration ratio. Redundant prefetches can overload the memory subsystem and thereby adversely affect the performance, and prefetch too much in advance can also result in cache pollution and performance degradation [11]. Prefetches are interesting only when the iteration trip count is large enough to make data access at the prefetch distance. This implies that for medium iteration numbers, prefetch instructions can be removed.

5 Experiments

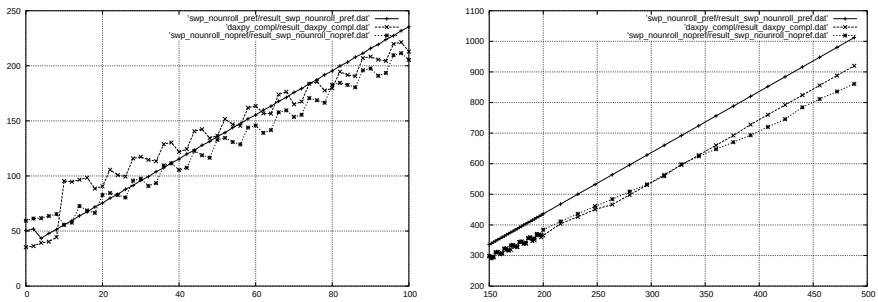
We consider three benchmarks: a `DAXPY` loop ($Y[i] = \alpha \times X[i] + Y[i]$) and two benchmarks from the CFP2000: `GALGEL` and `MGRID`. The `DAXPY` illustrates the combination of both unrolling and prefetch specialization.

`DAXPY`: Prefetch instructions must be generated for both `X` and `Y` arrays. It appears, that using prefetch degrades the initiation interval of the software pipelined loop due to extra pressure on memory slot.

Based on our performance model, there are three versions of the initial code:

- First zone: peeling, each block of 8 iterations costs $30 + N \bmod 9$. Peeling degree is set to 8 since is corresponds to the minimal latency in the DDG (4 cycles) which is just enough to schedule 8 floating point instructions.
- Second zone: disable prefetch, the formula is in $1 \times N + \alpha$[1]
- Third zone: enable prefetch, formula: $2 \times N + \alpha$

From the prefetch version, we know that the prefetch distance is set to 800B. Therefore, considering that every iteration is consuming 8 Bytes, it means that the loop needs to iterate at least 100 times before accessing the first prefetched data. So for this 100 first elements, it can used a loop without prefetch instructions. Performance results are detailed in Figure 3.



(a) Short iteration count for composed versions (b) Asymptotic behavior for composed versions

Fig. 3. (a) is a close-up of the relative performance between composed versioning, prefetch and no prefetch. For the first 8 iterations, composed versioning (referred as *DAXPY composed peeling/no prefetch* on the graph) outperforms all the other versions. However for the 9th iteration, composed versioning suffers from the overhead of filling up pipeline, while they are already filled-up for both other versions. This is consistent with our policy: overheads should be postponed as far as possible. Therefore even if these overheads still account for the same number of cycles their *relative cost* is smaller. Notice, that clearly composed versioning follows the same behavior than no prefetch version. In this example we chose to stick with no pretech up to 128 iterations. (b) details the behavior for large number of iterations. Composed versioning sticks with the no prefetch slope outperforming the prefetch version up to a hundred iterations. Beyond, it sticks to the original version, for the best asymptotic performance.

`GALGEL, loop b1_20`: The loop *b1_20* is a pipelined loop, and is one of the many versions generated by the compiler for one source loop. The loop has 8

[1] The Itanium architecture can not sustain one branch per cycle without inserting a stall cycles, the real formula is $1.7 \times N + \alpha$

iterations in the `train` input data set, 11 in the `ref` input data set. For this loop, we performed a peeling of one iteration. Table 4-(a) sums up the results of the peeling transformation.

Iteration count	Cycles		Performance Model		Gain
	Orig.	Peeling	Orig.	Peeling	
8	37760118	33984118	16xN+32	16xN+28	2.5 %
11	373744918	344995318	16xN+32	16xN+28	1.92 %

Fig. 4. Peeling one iteration out of loop b1_20, for each iteration count are given: the cycle count of the original loop and of the peeled loop (excluding peeled iteration), the cycle count according to a static performance model, and the performance gain of the peeling in % w.r.t. the original version.

`MGRID, loop b7_81`: The loop *b7_81* in the assembly code is memory access intensive, since it performs in two cycles two load-pairs (equivalent to four loads) and four stores. The loop uses one prefetch instruction and is pipelined. Peeling the loop does not bring significant performance gain, according to the performance model. Indeed each peeled iteration takes 2 cycles and interleaving peeled iterations does not reduce this latency. Therefore, as soon as the loop trip count exceeds the number of iterations peeled off the loop, the cycle count of the optimized loop should be similar to the cycle count of original loop.

As for prefetching, we split the loop into a sequence of two similar loops, the first without any prefetch instruction. The histogram of loop trip counts, provided by MAQAO [5] and presented in Figure 4-(b) shows that the loop trip counts are small enough to make prefetch useless.

Indeed, by removing prefetches in this single loop, the performance gain obtained for the whole benchmark is 25%. This illustrates a case where prefetches are counter-performant and trashes the data cache.

6 Conclusion

The stem of our work is the diagnosis that in scientific computing a consequent fraction of execution time is spent in loops with a small number of iterations. However, even modern compilers seem to bet everything on asymptotic performance. Clearly there are performance opportunities for non-asymptotic behaviors and optimization must be adapted to the size of data, and for loop, to the iteration range.

Therefore, we come out with a novel method to version codes. This compositional versioning limits the overhead, reduces costly decision tree height and exploits and executes as much as possible of the generated code. This new technique is based on loop versioning, according to the iteration count distribution. This is a generalization of simple asymptotic evaluations. Given a loop count distribution, either coming from static analysis of the code, provided by the user

through pragmas, or observed by profiling, we propose a smart loop versioning scheme. In particular, we split index sets so that each iteration range can be optimized more aggressively. The proposed optimizations are, for short range: peeling and for medium range: turning prefetching off, in addition to any versions proposed by the compiler. The first results on SPEC benchmarks show up to 25% speed up for one benchmark.

From an implementation point of view, our work is still in progress and while we are currently able to handle limited pieces of code and vector loops, we are now building the infrastructure to address the whole SPEC benchmark. One of the main issue to address is the switching overhead. In order to reduce it, we are investigating a way to peel off not only complete iteration but also software pipeline prologue and epilogue. Where the goal is to reschedule and interleave all these instructions allowing to switch directly from one version to a fully loaded pipeline.

References

1. Schwiegelshohn, U., Gasperoni, F., Ebcioglu, K.: On Optimal Parallelization of Arbitrary Loops. Journal of Parallel and Distributed Computing **11** (1991) 130–134
2. Darte, A., Robert, Y.: Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. Journal of Parallel and Distributed Computing **29**(1) (1995) 43–59
3. Rau, B.R.: Iterative modulo scheduling: an algorithm for software pipelining loops. In: Int. Symp. on Microarchitecture, San Jose, California, United States, ACM Press (1994) 63–74
4. Griebl, M., Feautrier, P., Lengauer, C.: Index set splitting. Int. Journal of Parallel Programming **28**(6) (2000) 607–631
5. Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.T., Jalby, W.: Exploring application performance: a new tool for a static / dynamic approach. In: LACSI Los Alamos Computer Science Institute Symposium. (2005)
6. Cooper, K., Dasgupta, A., Kennedy, K.: Vizer: A system to vectorize intel x86 binaries. In: LACSI Los Alamos Computer Science Institute Symposium. (December 2002)
7. Merten, M., Thiems, M.: An overview of the IMPACT x86 binary reoptimization framework. Technical report (July 1998)
8. Larus, J., Schnarr, E.: EEL: Machine-independent executable editing. In: Int. Conf. on Programming Language Design and Implementation. (1995) 291–300
9. McNairy, C., Soltis, D.: Itanium 2 processor microarchitecture. IEEE Micro **23**(2) (2003) 44–55
10. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Computing Surveys **27**(3) (1995) 367–432
11. Doshi, G., Krishnaiyer, R., Muthukumar, K.: Optimizing software data prefetches with rotating registers. In: Int. Conf. on Parallel Architectures and Compilation Techniques, Barcelona, Catalunya, Spain, IEEE Computer Society Press (sept 2001)