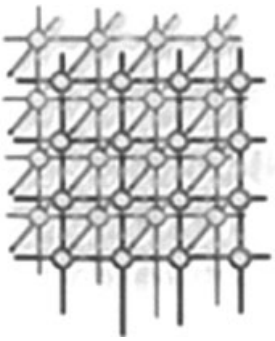
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:71–84
Published online 6 June 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1337



Compositional approach applied to loop specialization

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

University of Versailles, Versailles, France

SUMMARY

An optimizing compiler cannot generate one best code pattern for all input data. There is no 'one optimization fits all' inputs. To attain high performance for a large range of inputs, it is therefore desirable to resort to some kind of specialization. Data specialization significantly improves the performance delivered by the compiler-generated codes. Specialization is, however, limited by code expansion and introduces a time overhead for the selection of the appropriate version. We propose a new method to specialize the code at the assembly level for loop structures. Our specialization scheme focuses on different ranges of loop trip count and combines 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 than each version on its specific iteration interval. We illustrate the benefit of our method on the SPEC benchmarks with detailed experimental results. Copyright © 2008 John Wiley & Sons, Ltd.

KEY WORDS: loop optimization; code specialization; instruction level parallelism; instruction scheduling

1. INTRODUCTION

An optimizing compiler has a hard time to generate a code that will perform at top speed for an arbitrary data set size. In general, Schwiegelshohn *et al.* [1] have shown that 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] and tiling iteration domains are well-known techniques that improve latency. These transformations are driven according to the 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, whereas very large loop trip counts will favor deep software pipelining. To some extent, the code generated has to be specialized depending upon the data set size ranges and then has to use extensive versioning to apply these different specialized versions. The classical drawback of

*Correspondence to: D. Barthou, Laboratoire PRISM, Universit de Versailles St Quentin, 45 avenue des Etats Unis, F-78035 Versailles Cedex, France.
†E-mail: denis.barthou@prism.uvsq.fr

such an optimization scheme is code expansion. Moreover, when a code uses multiple specialized versions, some additional time is spent for the selection of the appropriate specialized code. This overhead has to be compensated by the speedup of the specialized version. Both drawbacks put a hard limit on the total number of different specialized versions that can be generated.

We propose, for loop structures, a new method to specialize the 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 numbers. We combine all these versions into a code that switches smoothly from one to the other while the iteration count increases. An important feature of our loop specialization is that it takes into account the overhead time to switch from one version to the other and computes the iteration count ranges for each version.

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 detailed experimental results.

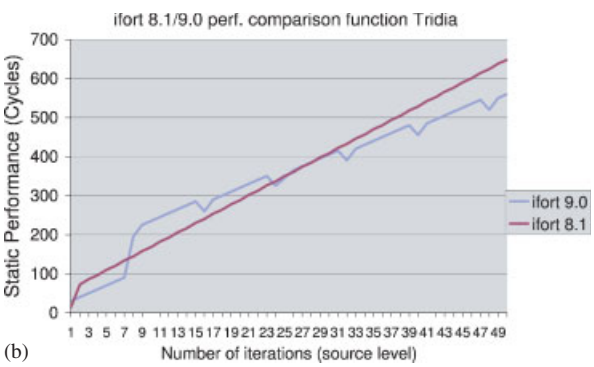
1.1. Motivating example: compiler and loop trip count heterogeneity

Loop optimization is a critical part in the compiler optimization chain. A routinely stated rule is that 90% of the execution time is spent on 10% part of the code. Another rule, implicitly used by the community, is that the number of iterations for loops in scientific code is *large*. Consequently, loops are almost systematically unrolled, deeply pipelined and prefetching is aggressively used to anticipated future data needs.

However, optimizations for asymptotic behavior involve some risk. For instance, in software pipeline, depth is always increased if it can reduce the initiation interval. This yields codes that 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 loop two times and generates a software pipeline of depth 2, whereas ICC 9.0 unrolls this loop 8 times and then applies software pipeline. The corresponding tail code is also software pipelined.

```
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))*CP1
1   CONTINUE
```

(a)



(b)

Figure 1. (a) Tridia Fortran source code and (b) performance of two assembly codes generated from the Tridia code (static performance measurement). 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.

DOI: 10.1002/cpe

The corresponding performance evaluation is

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

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

Overall, a code generation strategy based on asymptotic performance is a bet. This bet can pay off depending on how frequently hot loops exhibit a large number of iterations. In other words: up to which point large trip counts are dominant compared with short or moderate number of iterations?

To evaluate the importance of short loops, we perform a set of measurements on the SPEC FP 2000 benchmarks. Using the 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 captured by our tool. Nevertheless, ICC performs a very good job at generating counted loops especially for vector-like control structures.

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 loop iterations.

Figure 2 shows that the SPEC benchmarks spend at least 70% of their execution time in loops. Figure 3 details measurements made on the number of loop iterations for CFP2000 codes. The answer is surprising: 25% of the loop time is spent on loops with less than 8 iterations. A more detailed analysis shows that 6 over 14 benchmarks from the CFP2000 spend half of their loop time in loops with less than 16 iterations.

Therefore, loop tuning based on an infinite number of iterations misses the real performance opportunities and can easily lead to performance slowdowns. Indeed, many optimizations improve the performance if and only if the iteration count is large enough, for instance, prefetching loads in

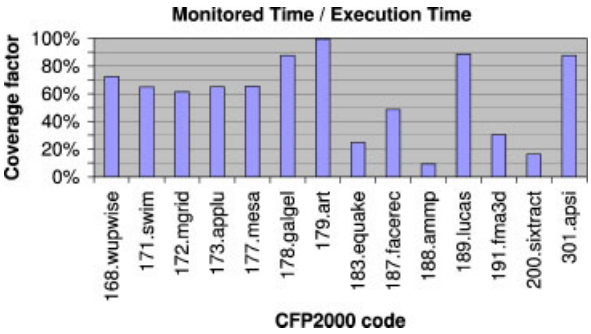


Figure 2. A fraction of the SPEC benchmark execution time spent on monitored loops. This is an indicator of the 'loopy' nature of the Spec benchmarks. Note that this is a conservative estimation as not all loops are monitored (innermost loop with only some constraints on control flow).

DOI: 10.1002/cpe

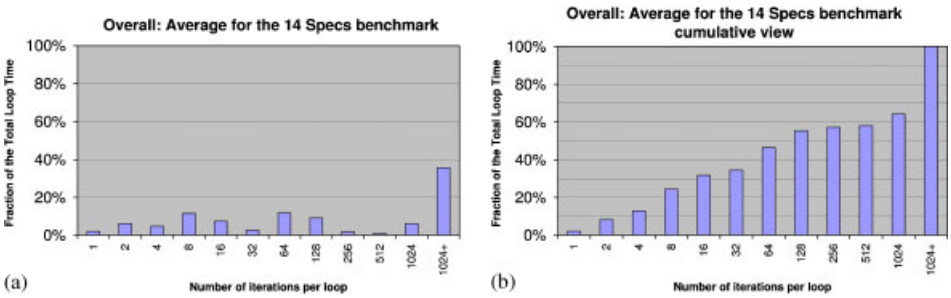


Figure 3. Loop execution time depending on the loops' number of iterations. Each graph depicts the execution time distribution per loop depending on the total number of iterations. Histograms summarize iteration weights for an interval in the power of two:]0, 1],]1, 2],]2, 4],]4, 8] and so on. For instance, the bar labeled 64 coalesces all loops with a total number of iterations between]33, 64]. These two figures summarize results obtained on the whole SPEC FP benchmarks suite. (a) Average on a per benchmark basis, i.e. each benchmark is considered individually without weight related to its execution time. (b) The same data but in a cumulative histogram, which is convenient to obtain the slope of the loop distribution. (a) SPEC FP overall loop distribution and (b) SPEC FP overall cumulative loop distribution.

advance data that will be used in some iterations later. If the iteration count is too small, prefetching only trashes caches and consumes memory bandwidth. Similarly, software pipeline on an Itanium architecture uses rotating registers and predication. Prolog, main loop and epilog are all in the same loop. Large pipeline depth has a high warming cost that is not compatible with small iteration counts.

Considering that small loops do not impact performance is missing the fact that these loops can be called many times and they cannot always be fully unrolled (if iteration count is not known statically). Short loops do matter. Hence, this advocates for an optimization technique that combines (or compose) different versions of these loops, optimized for different ranges. The compositional approach proposed in this paper is a method to deliver high performance for both small and large numbers of iterations. The key idea is to specialize codes in a vertical manner; executed codes differ when the number of already executed iterations exceeds a threshold. In some sense, this can be seen as the code accompanying the data.

2. COMPOSITIONAL APPROACH

The idea of the approach, given different codes (and schedules) for the same loop, is to combine or compose them into one code that achieves the same performance as the best code, for any iteration count.

2.1. Iteration ranges and optimizations

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 they apply.

DOI: 10.1002/cpe

1. *Peeling*: Peeling enables rescheduling of the first iterations of a loop. It generates more opportunities for a better resource usage, with a free schedule, at the cost 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 the 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 better ILP, therefore increasing the 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 starts of iterations. This comes at the cost of the latency required to execute a complete iteration (MAKESPAN), 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.

The classical approach for multiversioned code is first to decide *a priori* when to switch from one version to the other and then to create a decision tree or a table of functions to select the appropriate code version for a given parameter value (here, loop iteration count). However, for small loops, the execution time of the loop can be as low as a few dozen of cycles. The overhead of the decision procedure (decision tree or table of functions) may be too high for these loops, outbalancing the advantages of a specialized version. The following section presents a performance model that takes into account this overhead.

2.2. Performance model

The main principle of the compositional approach and the model proposed here is as follows. Instead of paying the cost of an overhead to select a code for short loops or a code for large loops, we propose to start with the short loop code and then, if the loop has more iterations, switch to a code adapted for larger iteration count. Therefore, the overhead, due to the transition between the two versions, is paid only if the loop has a large enough iteration count.

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 $c_1(i) = \alpha_1.i + \beta_1$, where α_1 is a rational number, β_1 an integer and the cycle count is rounded off. Similarly, for L_2, $c_2(i) = \alpha_2.i + \beta_2$. For instance, in Figure 1, the cycle count of loop L_1 generated by ICC 8.1 is defined with $\alpha_1 = 12$ and $\beta_1 = 48$ 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 $i < \beta_2 - \beta_1/\alpha_1 - \alpha_2$ and L_2 outperforms L_1 for larger iterations.

We would like to build a *best* code such that

$$\forall n, c_{best}(n) = \min_k(c_k(n))$$

DOI: 10.1002/cpe

This *best* code is built by an optimization function min: $\min(L_1, L_2) = best$. The function min defines a minimum of codes with respect to performance for all iteration values. Owing to the difficulty in 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 min loop:

1. The asymptotic performance (in cycle/iteration, when the iteration count grows) is the same as the best asymptotic performance of L_1 and L_2.
2. Each loop is possibly called many times, each time with a possibly different loop trip count. For a given distribution of loop counts, the average gain in cycle/iteration compared with the best asymptotic performance of L_1 and L_2 is positive.
3. When performance of both loops L_1 and L_2 are the same, the loop selection is done according to the best 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 executions of the loop, some cycles have been gained. The reason is that some overhead may appear when executing from one version to the other. The best code would have the cycle count as follows:

$$c_{12}(i) = \begin{cases} i \leq B\text{: } c_1(i) \\ i > B\text{: } c_2(i) - c_2(B) + c_1(B) + \gamma \end{cases}$$

where B is the integer $\beta_2 - \beta_1/\alpha_1 - \alpha_2$ and γ represents the overhead necessary when going from one version to the other. This overhead represents register initializations, branch mispredicts, etc.

The difference in cycles/iterations between the asymptotic best loop and the new loop $\min(L_1, L_2)$ is, for an iteration count i, obtained as

$$dpi(i) = \begin{cases} i \leq B\text{: } (c_2(i) - c_1(i))/i \\ i > B\text{: } (c_2(B) - c_1(B) - \gamma)/i \end{cases}$$

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 $[1 \ldots N]$, 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, the distribution of values obtained 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 on average faster than the best asymptotic loop L_2. This value is positive when $N < B$ as each difference/iteration is positive for all iterations $i < B$. For higher values of N, $adpi(N) > 0$ if

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

DOI: 10.1002/cpe

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

$$\gamma < c_2(B) - c_1(B)$$

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 $c(L_1)$ and $c(L_2)$ 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 with respect to 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.

3. ASSEMBLY-TO-ASSEMBLY TRANSFORMATION

We first present an assembly code dependence analysis and then describe two particular transformations, loop peeling and transformation of prefetching an Itanium architecture, but we believe that this can be generalized to other platforms.

3.1. Code flattening and dependence graph

Code transformations preserve code semantics when they preserve data dependences. The *data dependence graph* (DDG) of the assembly codes is built with the following method. 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 [6]. Loop flattening is a transformation that removes the effect of software pipeline: it carefully renames registers according to their rotating status or not and predicates are used to retrieve the iteration number when the instruction becomes valid.

A DDG 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.

DOI: 10.1002/cpe

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 in that manner.

3.2. Peeling and prefetching transformations

We show the composition approach using two transformations: 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 to match two different bounds of two subsequent loops. Generally, the positive effect of this technique is better understood if explained in conjunction with a loop fusion. In our approach, peeling also has a positive effect if explained in conjunction with software pipelining. Compared with warming up stages of a 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 are 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 a 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; therefore, the latency α' is less than 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. A 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 a 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. $\mathrm{Log}_2(N)$ comparisons and $\log_2(N)/6$ cycles are necessary to set the predicate registers of N peeled iterations. Thus, the overhead is limited to a couple of cycles.

For prefetching, the prefetch distance is computed 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 one, performing the remaining iterations.

DOI: 10.1002/cpe

3.3. Robustness, overhead and code size

Robustness: Compositional versioning can be applied even when the loop trip count is not known in advance (for instance, a while loop). These kinds of loops are especially difficult to optimize with classic versioning techniques. In fact, compositional versioning can be applied to a loop with a complex control flow (for instance, with a break in the loop body). These characteristics make compositional versioning a robust optimization.

Overhead: The weakness of code specialization is that the cost of the decision tree has to be paid for all cases. This is especially harmful when branching to the version optimized for a small number of iterations (for which every cycle is important). On the other hand, in compositional versioning, the overhead appears only when the code switches from one version to the other. Hence, there is no branch to the version optimized for a small number of iterations; this is the natural entry point of the loop body. The overhead is paid only when the trip count exceeds a threshold and leads to the activation of another version of the code.

Code size: *Traditional versioning*: The overall code size corresponds to the number of versions plus a decision tree (to select the most appropriate version at runtime). N being the total number of versions, the decision tree can be estimated as $\log(N)$ compare instructions.

Compositional versioning: The code size corresponds to the number of versions plus an extra glue stage to ensure correct transition from one version to the other. For N version we need $N-1$ glue stages. At worst, the cost of the glue stage is one compare and one mov instructions (to connect the output of one preceding version to the input of the next version) per output value of each version. Considering the Itanium issue abilities, each glue stage cost $\frac{1}{3}$ of a cycle per output value of a version.

Therefore, compositional versioning comes at the cost of a slightly higher code size than the traditional loop specialization.

4. RELATED WORK

Specialization is a well-known technique to obtain high-performance programs [7]. Compiled time specialization often boils down to the generation of codes that are in mutually exclusive execution paths. Splitting iteration space 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 (changing the cost of memory latency). However, very few works resort to loop versioning to explicitly reduce the overall latency of the loop. This is due to the intractability of general performance models (finding the best latency affine schedules is still a difficult problem). This 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 the performance of inner loops accurately. This is easier on assembly codes than on the source code.

Software pipelining [8] 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

DOI: 10.1002/cpe

presence of data dependencies, which greatly improves the 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 counts. As it comes at the cost of the code size, compiler heuristics usually prefer not to 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 as the distribution is then more accurate.

Prefetch works by bringing in data from memory well before they are needed by a memory operation. This hides the cache miss latencies for data accesses and thus improves the performance. Typically, when a cache line is brought in from the 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 at every iteration, it may degrade the 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.

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.

About the cost of switching from one version to the other, several techniques are possible: cascade of branch instructions forming a tree or a jump-table. Jump-table has the advantage of a fixed overhead and was consequently selected by Diniz and Rinard [9] to allow the code to switch from one version to the other according to the dynamic behavior (in their article, versions correspond to different parallelization schemes). However, we have thoroughly analyzed the cost of branch in [10] and, due to the relatively high latency of a calculated branch, for a limited number of versions a decision tree made of branch instructions is more efficient on the Itanium architecture.

5. EXPERIMENTS

We consider four benchmarks: three real programs, of which two belong to the CFP2000: `GALGEL` and `MGRID`. The third benchmark is BLAST, the code of reference for searching genomic sequences alignment. The last code is the simple `DAXPY` loop ($Y[i] = \alpha \times X[i] + Y[i]$), where its main purpose is to illustrate the combination of both unrolling and prefetch specialization.

5.1. Code generation

These experiments were conducted in a semi-automated manner. Considering a given loop in the assembly code (produced by the compiler), MAQAO is able to un-pipeline the loop, detect independent instructions and isolate a single iteration. When peeling is applied to several iterations, some

DOI: 10.1002/cpe

human intervention is required to interleave these iterations at the assembly level. The correctness of the final code is then checked by the DDG computation performed by MAQAO.

5.2. 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 the extra pressure on the memory. As a matter of fact, on Itanium only a limited number of memory instructions can be issued per cycle (up to 4). If for a given cycle, 4 memory instructions are already scheduled, to add a prefetch the compiler has to re-do a schedule on 2 cycles instead of 1.

On the basis of our performance model, there are three versions of the initial code:

- First zone: Peeling, each block of 8 iterations costs $30 + N \mod 9$. Peeling degree is set to 8 as it 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 $1 \times N + \alpha$[‡].
- Third zone: Enable prefetch, the formula is $2 \times N + \alpha$.

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. Therefore, for these 100 first elements, it can use a loop without prefetch instructions. Performance results are detailed in Figure 4.

5.3. 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 iterations in the `train` input data set, 11 in the `ref` input data set. For this loop, we performed peeling off of one iteration. Figure 5(a) summarizes the results of the peeling transformation.

5.4. MGRID, loop b7_81

The loop *b7_81* in the assembly code is memory access intensive, as 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 a 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 the 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 the loop trip counts, provided by MAQAO [5] and presented in Figure 5(b), shows that the loop trip counts are low enough to make prefetch useless.

[‡]The Itanium architecture cannot sustain one branch per cycle without inserting a stall cycles; the real formula is $1.7 \times N + \alpha$.

DOI: 10.1002/cpe

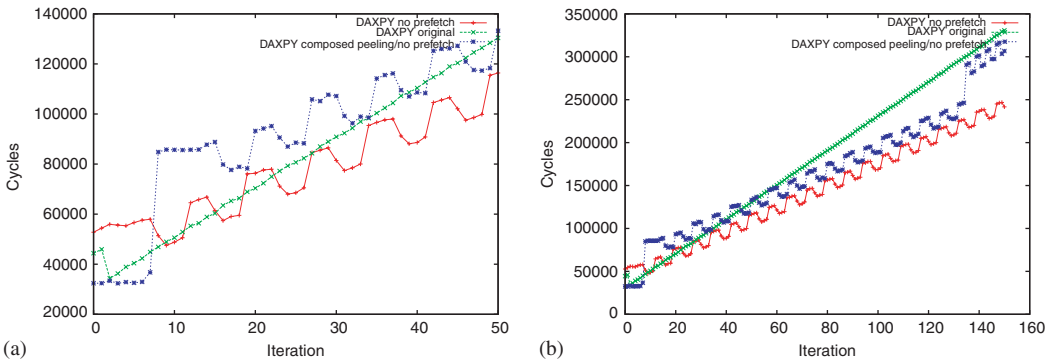


Figure 4. (a) A close-up of the relative performance between composed versioning, prefetch and no prefetch. It appears that for the first 8 iterations, composed versioning (referred to as *daxpy_compl* on the graph) outperforms all the other versions. However, for the 9th iteration, composed versioning suffers from the overhead of filling up pipeline, while they are already filled up for the two 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. Note clearly that composed versioning follows the same behavior as the no-prefetch versioning. In this example, we chose to stick with no pretech up to 256 iterations (short iteration count for composed versions). (b) The behavior for a large number of iterations. Composed versioning sticks with the no-prefetch slope outperforming the prefetch version up to a hundred iterations. In addition, it sticks to the original version, which is the best asymptotic version compared with the no-prefetch version (asymptotic behavior for composed versions).

Iteration count	Cycles		Performance Model		global gain	local gain
	Orig.	Peeling	Orig.	Peeling		
8	37,760,118	33,984,118	16xN+32	16xN+28	2.5 %	10.0 %
11	373,744,918	344,995,318	16xN+32	16xN+28	1.92	7.7 %

(a)

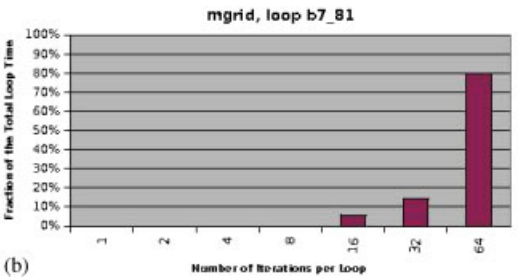


Figure 5. (a) Peeling one iteration out of loop *b1_20*. For each iteration counts are given: the cycle count of the original loop and the peeled loop (excluding peeled iteration), the cycle count according to a static performance model and the performance gain of the peeling in % w.r.t. the original version. (b) `MGRID` loop *b7_81* execution time depending on the number of iterations. The histogram summarizes iteration weights for an interval in the power of two:]0, 1],]1, 2],]2, 4],]4, 8] and so on.

DOI: 10.1002/cpe

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 trash the data cache.

5.5. BLAST

This code is the reference for the search of sequences alignment. We use the ncbi 2006/04/20 version, which is composed of 1480 source files, which means 2.7 million lines of C code. Three hot functions can be isolated in this code, of which two exhibit interesting short loops characteristics.

- *BlastAaWordFinder_TwoHit*, fraction of the execution time: 38.4%. A very large number of iterations.
- *BlastAaScanSubject*, fraction of the execution time: 18%. A hot loop with a number of iterations less than 64.
- *BlastAaExtendTwoHit*, fraction of the execution time: 15.1%. A hot loop with a number of iterations mostly equal to 3.

For the hot loop of *BlastAaScanSubject*, we proceed with the peeling off of the first two iterations. In the case of the hot loop in *BlastAaExtendTwoHit*, we proceed also with peeling off of the first iteration but we also use a version without prefetch. The limited number of iterations hints that the prefetch efficiency may be low. Table I summarizes the gains observed on Blast, global gains at the scale of the whole gain and their local meaning, i.e. improvement for the execution time of the optimized loop. Overall Blast was accelerated by 2.5%.

5.6. Discussion

Experimental results clearly show the potential of our compositional approach; however, it remains a niche. The main limitation remains in the number of optimizations available for composing versions. Performance improvements are not always brought by a more suitable prefetching or by peeling off few iterations. Vectorization (with SSE on IA32 or multimedia instruction on IA64) should be handled to provide more opportunities. An additional problem is the sensitiveness to the quality of the versions to compose. If the compiler fails to generate a relevant code, composing under-optimal versions will not deliver relevant performance.

Table I. Compositional approach gains and short loop specialization applied to the Blast code.

Original Blast	opt. BlastAaExtendTwoHit	Global gain	Local gain
57.61	57.16	0.8%	5.3%
	opt. BlastAaScanSubject	Global gain	Local gain
	56.60	1.7%	9.3%
Combined gain	56.15	+2.5%	

Time is given in billion of cpu cycles.

DOI: 10.1002/cpe

6. CONCLUSION

The stem of our work is the diagnosis that in scientific computing a consequent fraction of execution time is spent on 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 the loop, to the iteration range.

Therefore, we have developed a novel method for version codes. This compositional versioning limits the overhead due to version code selection 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 the index sets so that each iteration range can be optimized more aggressively. The proposed optimizations are for a short range: peeling and for a medium range: turning prefetching off, in addition to any versions proposed by the compiler. The first results on the SPEC benchmarks show up to 25% speedup for one benchmark.

From an implementation point of view, our work is still in progress and although we are currently able to handle limited pieces of code and vector loops, we are building the infrastructure to address the whole SPEC benchmark. One of the main issues to address is the switching overhead. In order to reduce it, we are investigating a manner 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* 1991; **11**: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* 1995; **29.1**:43–59.
3. Ramakrishna Rau B. Iterative modulo scheduling: An algorithm for software pipelining loops. *Proceedings of the International Symposium on Microarchitecture*, San Jose, CA, 1994; 63–74.
4. Griebl M, Feautrier P, Lengauer C. Index set splitting. *Journal of Parallel Programming* 2000; **28.6**: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. *Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, 2005.
6. Intel Itanium2 Processor Reference Manual for Software Development and Optimization. http://download.intel.com/design/Itanium2/manuals/25111003.pdf [May 2004].
7. Sias JW, Ueng S-Z, Kent GA, Steiner IM, Nysom EM, Hwu W-M. Field-testing IMPACT EPIC research results on Itanium 2. *Proceedings of the Annual Symposium on Computer Architecture*, *ISCA*, Munchen, Germany, 2004.
8. Allan VH, Jones RB, Lee RM, Allan SJ. Software pipelining. *ACM Computing Surveys* 1995; **27.3**:367–432.
9. Diniz PC, Rinard MC. Dynamic feedback: An effective technique for adaptive computing. *Proceedings of PLDI*, Las Vegas, NV, 1997; 71–84.
10. Carribault P, Lemuet C, Acquaviva J-T, Cohen A, Jalby W. Branch strategies to optimize decision trees for wide-issue architectures. *Languages and Compilers for Performance Computing* (*Lecture Notes in Computer Science*). Springer: Berlin, 2004.

DOI: 10.1002/cpe