Optimizing Code Through Iterative Specialization

Minhaj Ahmad Khan
Univ. of Versailles
France
mik@uvsq.fr

Henri-Pierre Charles
Univ. of Versailles
France
hpc@uvsq.fr

Denis Barthou
Univ. of Versailles
France
denis.barthou@uvsq.fr

ABSTRACT

Code specialization is a way to obtain significant improvement in the performance of an application. It works by exposing values of different parameters in source code. The availability of these specialized values enables the compilers to generate better optimized code. Although most of the efficient source code implementations contain specialized code to benefit from these optimizations, the real impact of specialization may however vary depending upon the value of the specializing parameter.

In this paper, we suggest the specialization of code to acquire an iterative approach. For some specialized code, we search for a better version of code by re-specializing the code, followed by a low-level code analysis. The specialized versions fulfilling the required criteria are then transformed to generate another equivalent version of the original specialized code.

The approach has been tested on Itanium-II architecture using *icc* compiler. The results show significant improvement in the performance of different behchmarks.

Keywords

Compiling techniques, program optimization and specialization, analysis and transformation, programming languages implementation

1. INTRODUCTION

The specialization of code [4] is a technique that makes values of different variables available to the compiler. Having a better knowledge of values that variables can take, enables the compiler to generate better code. This principle drives many well-known optimizations: from partial evaluation to profile value-driven transformations or the runtime optimizations. When a variable does not change its value and behaves like constant for a set of instructions, then the optimizations such as constant propagation, dead-code elimination, strength reduction or use of machine idioms can all directly take advantage of this information. This infomation may also be beneficial for analyses that may in turn enable complex optimizations. For instance, constant loop bounds may trigger loop unrolling, fusion and computation of prefetch distance. This is why, the highly optimized libraries such as ATLAS [13] and FFTW [5] also contain specialized code.

With the specialization, there are many optimizations which are dependent on the value of the specializing variable. For example, cyclic/acyclic scheduling with region enlarging or compaction optimizations[14], modulo scheduling/pipelining [11] with different II (Initiation Interval[1]) & number of stages and different unrolling factors, all can largely impact the execution speed of an application. Similarly, while scheduling instructions, the amount of parallelism is directly or indirectly dependent on the code size [14]. This can be achieved through code specialization which largely impacts code size and thereby may modify the sequence of optimizations. In most of the cases, it is not apparent that specialization with a particular value would bring the best performance for application execution. This fact makes the iterative compilation approach [3] inevitable to search for a better specialized version.

In this article, we present a new method based on an iterative approach that is aimed at improving the performance of specialized code. It works by re-specializing code, performing code validation and transforming the object code to make it equivalent to the original code.

2. PRINCIPLE OF ITERATIVE SPECIALIZATION

The iterative specialization relies on the fact that after code specialization, the compiler produces code that can be divided into *groups* of versions such that within a *group*, the optimizations (including register allocation and scheduling) are similar. Therefore, within one *group* (a set of versions of the specialized parameter), all the code versions have the same instructions and may only differ by some immediate constants [1][9].

To see the impact of different specializing values, consider the DAXPY code given in Figure 1 which is used to calculate $Y = Y + alpha * X$, where X and Y are vectors and $alpha$ is a constant value.

When this code is compiled, analysis of assembly code shows that different optimizations are performed for differ-

[1] In pipelined loop, number of cycles required to start new iteration

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

```
#define size 7
void DAXPY(double * X, double * Y, double alpha)
{
    for (int i=0; i < size; i++)
    Y[i] = Y[i] + alpha * X[i] ;
}
```

Figure 1: Sample DAXPY code

ent values of the variable *size*. When the variable *size* is specialized to be 7, the compiler performs multi-versioning thereby generating two versions at the object code level. In contrast, the code specialized with 15 is unrolled and also pipelined with Initiation Intervals of 2 and 4 cycles for its two versions. The code similar to one specialized with 15 is also generated after specialization of *size* variable with value of 13. The assembly versions generated after specialization with 13 and 15 differ in statements as given in Figure 2.

//*size*=13	// *size*=15
mov ar.lc=**12**	mov ar.lc=**14**
....	....
add r24=**48**,r14	add r24=**56**,r14
add r26=**48**,r18	add r26=**56**,r18

Figure 2: Assembly Code generated by icc v 9.1

It means that the versions for 7 and 13 fall in two different *groups*, whereas those for 13 and 15 fall in the same *group*. For both *groups*, the compiler has performed different optimizations. If we represent the sequence of optimizations (scheduling, register allocation etc.) as patterns, the set of values {7,13,15,31,33,143} generates four *groups*, {7}, {13,15}, {31,33}, {143} as shown in Figure 3. Each *group* has different optimizations from other *groups*.

Within any two versions of a *group*, the constant values in object code are in fact functions of the specialized parameter in source code. These functions are either affine (i.e. *val**A+B, where A and B are constants and *val* is the value of the specialized parameter) or may be inferred from the expressions in the source code (found by static analysis of source code). We consider in this paper only regular codes w.r.t. the specialized parameters: these are neither involved in IF-condition predicates, nor in bounds of loops that are not alike Fortran DO-loops. The immediate operands that differ can be replaced by other valid immedidate values to produce another version which is equivalent to original specialized version. The instructions which need to be transformed, will be termed as *annotated* instructions. We select single version from the *group* as a template version which contains *annotated* instructions to be transformed. Since all the versions in a *group* are equivalent, any version can be selected as the template.

Before performing transformation, the template code needs to be validated against the values generated in both the cases i.e. affine and non-affine formulae. Each new value should fall within the limitations offered by the instruction set of the architecture.

Once validated, the old immediate values of each instruction of the template are replaced by new immediate values which are obtained by using the formulae (affine or non-affine) for each *annotated* instruction. This transformation actually makes the code equivalent to original specialized code. The original code exists in other *group* to benefit from different optimizations, but performs the same functionality as of original specialized code. Since we have more than one *group*, the specialization and transformation are performed iteratively to select faster version of code. Both the re-specialization of code and the transformation, are entirely performed at static compile time incurring no overhead during execution of the application.

2.1 Formal Context of Iterative Specialization

This section describes the context of iterative specialization together with notions of *groups* and templates.

We first define specialization as follows: Given a program P and a parameter *param* specialized with value *val*, specialization is a function that converts the program P to P_{val}^{param} where *param* is the name of the parameter having value *val* such that many operations related to *param* have been optimized. The specialization function can be given as:

$$Spec(P, param, val)\colon P \rightarrow P_{val}^{param} \quad (1)$$

The basic idea of iterative specialization is to generate n versions of the program $P, P_{val_i}^{param}$ using different functions $Spec_i$,

$$Spec_i(P, param, val_i)\colon P \rightarrow P_{val_i}^{param} \quad (2)$$

such that $val_i \neq val$ and $P_{val}^{param} \neq P_{val_i}^{param}$ $\forall 1 \leq i \leq n$.

Once, different versions are generated, we can search for different subsets of val_i called *groups* by generating versions $P_{val_i}^{param}$ such that:

$$(P_{val_j}^{param} - P_{val_k}^{param}) \vee (P_{val_k}^{param} - P_{val_j}^{param}) = Imm, \quad (3)$$

where Imm is a set of *Immediate Operands*(i.e. constants) and $val_j \neq val_k \neq val$ for $1 \leq j < k \leq n$, We annotate all such instructions and find the formulae on which these immediate operands are based. Any of these versions in the *group* may be used as template. Let $|T|$ be the number of the found formulae, then for each *annotated* instruction of the template (which differs from others in a *group*), we restrict each immediate operand to fall into any of two following categories:

1. Be an affine function of its specializing value. i.e. For a *group* G having a version P_v^{param} specialized with value v, We must have,

   $$Imm_G = A * v + B. \quad (4)$$

   The values of coefficients A and B can be obtained after solving the system of linear equations. In this case, the template instruction formula would be $A * val + B$ if each given *annotated* instruction is valid for the old parameter *param*. To validate an instruction, we proceed as follows.

   Let A_p and B_p be the coefficients calculated for p-th instruction, then we have,

   $$S_p = \left\lceil \frac{INSTMIN_p - B_p}{A_p} \right\rceil, E_p = \left\lfloor \frac{INSTMAX_p - B_p}{A_p} \right\rfloor,$$

   where $INSTMAX_p$ represents maximum value that can be used as immediate operand for p-th *annotated* instruction. The new valid range for the parameter *param* with S = $\{S_p$, for p=1 to $|T|\}$ and E=$\{E_p$, for p=1 to $|T|\}$, can be represented as:

   $$MAX(S) \leq param \leq MIN(E). \quad (5)$$

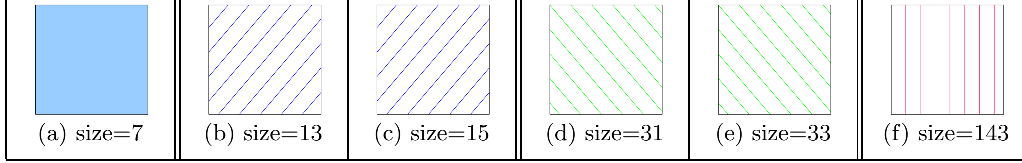


Figure 3: Different patterns of code due to different optimizations

2. Be a non-linear function whose formula can be directly derived from the *uses* of the specialized parameter in those expressions which only contain the parameter as *rvalue*. To ensure correctness, we require that such formula be similar in all the versions of the *group* G. For a version specialized with value v, let f be a function on which these values are based, then, we must have

$$f(v) = Imm_G, \tag{6}$$

where $Imm_G \in Imm$ is the immediate value (as found in Equation 3). Function f should be unique in these versions and would be the template instruction formula. To validate p-th instruction, we have $\forall\ p = 1\ to\ |T|$,

$$INSTMIN_p \geq f_p(val) \geq INSTMAX_p, \tag{7}$$

where $INSTMIN_p$ and $INSTMAX_p$ are respectively the minimum and maximum values that the architecture allows for p-th *annotated* instruction of the template.

If the code does not conform to above mentioned conditions (Equations 4-5 or 6-7), it is illegal for re-specialization, and, the assembly code would need to be specialized with other values for that parameter.

However, in the case where code conforms to the above given conditions, the next step is to transform all the instructions of the template. The new values are calculated by using the original value *val* for the parameter *param* in the corresponding template instruction formula. The performance of transformed code is then profiled and the entire process of re-specialization and transformation is iterated (within threshold time limit) for all values in the input interval.

2.2 Specialization Complexity

The main complexity of the algorithm lies in solving the system of equations followed by searching the *uses* of the specializing parameters. For single parameter, the complexity of such a system is $O(d(1+e))$, where we have d instructions *annotated* and search through the e expressions. However, for p parameters, the complexity increases to $O(d(p^3+e))$ at static compile time.

3. ISPEC: IMPLEMENTATION FRAMEWORK

The iterative specialization framework ISPEC is aimed at generating a better optimized version equivalent to the original specialized code. It therefore iteratively specializes input code (within threshold limit[2]), generates *groups*, checks the conditions of code correctness (as described in Section 2.1), and transforms the code to generate the equivalent version. It also performs instrumentation to profile the execution performance of the code.

[2]input configuration file to ISPEC Core contains threshold, mode and compilation parameters

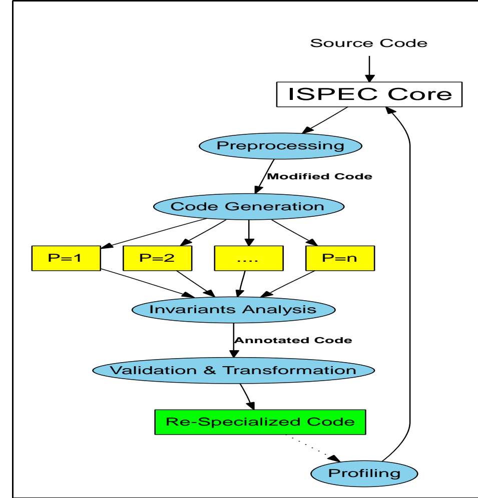


Figure 4: ISPEC Framework Architecture

The main phases of iterative specialization have been depicted in Figure 4.

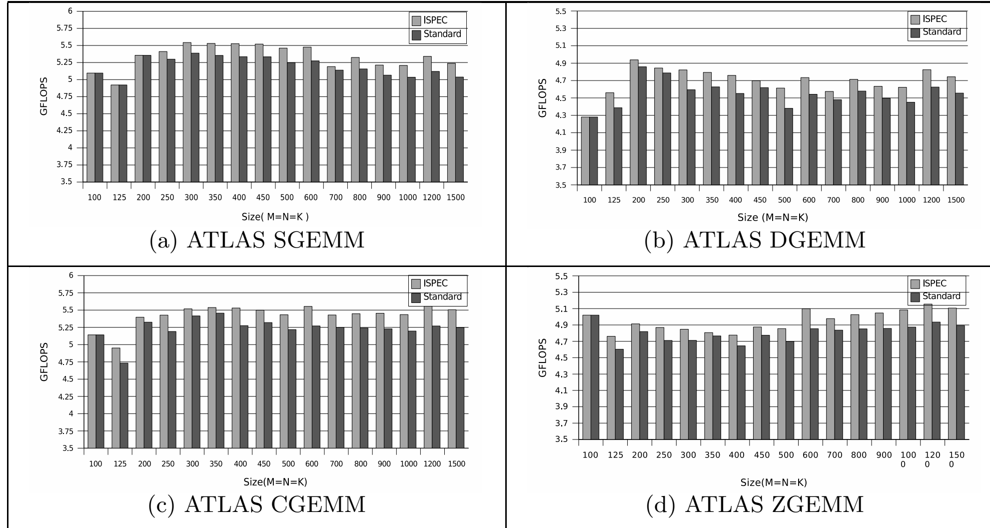
3.1 Preprocessing and Analysis

The preprocessing requires the user-defined directive of the following form to specify parameters and their interval of values for which specialization would proceed.

$$\#ispec\ Param_Name|LOOP\ startVal\ \ endVal$$

The source code is parsed by ISPEC core and modified either for the parameters defined using *#define* directive of C language, or for the loop counts, both should be preceded by the *#ispec* directive. Their values are replaced with new values as given in the interval. By default, ISPEC makes use of all values specified in the interval (ISPEC *Normal* mode), however, it also supports use of pre-defined prime numbers to reduce the time of iterative specialization (ISPEC *Quick* mode).

The code generation is accomplished by invoking the compiler to obtain different versions. The generated versions are compatible and can be transformed if they differ only in immediate constants. The exploration of *groups* proceeds by finding differences among object code versions at corresponding instructions in the code. The versions differing only in immediate constants are added to the *groups*, and first version from each *group* is selected as a template. The template is checked for both linear (by solving affine formulae) and non-linear cases (by searching *rvalues* of the expressions containing *uses* of specializing parameter). Consequently, the formulae are generated corresponding to each *annotated* instruction in the template.



(a) ATLAS SGEMM (b) ATLAS DGEMM

(c) ATLAS CGEMM (d) ATLAS ZGEMM

Figure 5: ATLAS Performance Results

3.2 Validation and Transformation of Code

For each instruction of the template, the new value is computed by putting original value of the parameter in the corresponding formula. The result value is then validated against the possible size of immediate operand for that instruction. If all the instructions are valid according to the criteria described in Section 2.1, the immediate operands of the *annotated* instructions of the template are replaced by new calculated values. This transformation actually generates another valid version for the considered regular codes (Section 2).

3.3 Profiling and Iterating Code Specialization

When all the *annotated* instructions in the object code template have been modified, the object code is instrumented (incorporating pfmon-lib) to profile the execution time of specialized code. For each value in the specified interval of the parameter, the entire procedure of searching equivalent code and transformation is iterated. The search space is limited by profiling execution within the threshold time limit which is set as configuration parameter.

4. EXPERIMENTAL SETUP AND RESULTS

The experiments have been performed over Intel Itanium-II (IA-64) with 1.5 GHz processor clock speed. The iterative specialization has been applied to different types of code in ATLAS-3.6.0, FFTW-3.0.1 and FFMPEG libraries using 1 hour threshold time limit. The code has been compiled using *icc v 9.1* with *-O3* optimization option.

4.1 ATLAS Library

ATLAS (Automatically Tuned Linear Algebra Software) [13] consists of portable routines to solve different mathematical problems.

The iterative specialization has been applied to linear algebra BLAS-3 matrix-matrix multiplication kernels (routines $a1_b1$ and $a1_bX$) for each of SGEMM, DGEMM, CGEMM and ZGEMM implementations. These routines use N, M and K as dimensions of input matrices. The new corresponding templates in all cases contained optimizations with better scheduling of code and data prefetching. The templates required 1, 32, 32 and 2 instructions for SGEMM, DGEMM, CGEMM and ZGEMM respectively to generate corresponding equivalent versions. The performance results of ATLAS routines are shown in GFLOPS in Figure 5. The iterative specialization is able to produce up to 8% speedup for these routines.

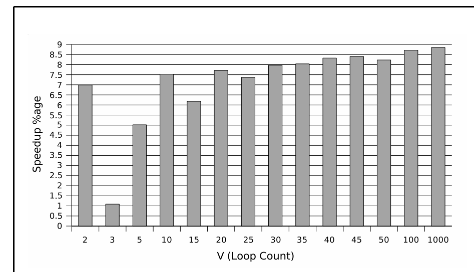


Figure 6: FFTW Performance Results

4.2 FFTW

FFTW (Fastest Fourier Transform in the West) library [5] is a set of specialized C routines called *codelets* which are used to calculate fourier transforms.

The *codelet* $st1_8_128$ (already specialized for input-output stride to be 128) is optimized through iterative specialization. The template differed from original specialized version in number of memory loads and register usage, and required 20 instructions to be transformed. The speedup obtained with different values of loop count (`v`) variable is shown in Figure 6.

4.3 FFMPEG

For multimedia applications, FFMPEG [7] contains a set of optimized libraries such as *libavcodec* which uses Discrete Cosine Transform (DCT) for conversion of images in differ-

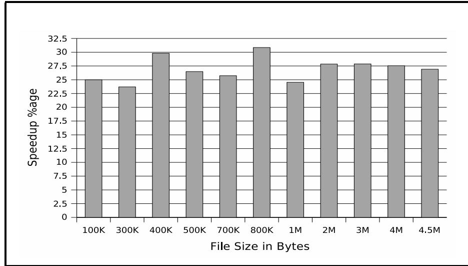


Figure 7: FFMPEG Performance Results

ent formats.

The template found after iterative specialization of $DCTSIZE$ parameter was very small as compared to original code due to global acyclic scheduling, and required 26 instructions of object code to be transformed. As shown in Figure 7, the iterative specialization is able to achieve speedup up to 31% for conversion of different `mpeg` files to raw format `yuv` for different sizes in FFMPEG library benchmark.

5. RELATED WORK

The iterative specialization approach is different from other iterative compilation approaches [13][12][3]. These approaches mainly search for best transformation/optimization parameters, whereas, our approach uses code specialization to obtain different optimizations and can be used as another parameter in parameter optimization space.

In work related to code specialization, the Tempo specializer [4] performs specialization both at static compile time and runtime. However, it generates different versions with specialized code thereby increasing the size of code with no guarantee of producing the best code after specialization due to the large overhead of heavy dynamic code generation.

The HySpec [1] specializer is able to generate code that is based on templates. This approach requires the template to be generated and optimized at static compile time. A limited set of binary instructions is then specialized at dynamic compile time. Although it incurs less overhead than complete code generation, its template may not be fully optimized as in our approach of iterative specialization. Similarly, an iterative specialization approach that is oriented towards improvement of multimedia applications has been given in [9]. However, it is restricted to DCT implementations with fixed input size of 8.

Other specializers and code generators like DyC [6], Dynamo [8] and Tick C[10] also optimize code during execution. The specialization is based on static and dynamic analyses but the optimizations are performed at runtime to generate specialized versions. These approaches therefore require the code to be invoked multiple times to amortize the overhead. Moreover, the increased object code size indirectly impacts the application performance due to cache penalties.

In contrast to these approaches, our specialization approach is iterative, and therefore suitable to be performed at static compile time. For the libraries, since the functions are expected to be executed many times, the overhead of iterating specialization at static compile time is less significant.

6. CONCLUSION

In this article, we have presented a method to fully exploit the feature of specialization that improves performance by generating the minimum object code requiring no runtime overhead. We search for equivalent versions of code, and generate different *groups* of versions where each *group* contains versions for which the code optimizations are similar after specialization. A version from a *group* is selected and transformed to generate an equivalent version of original code. This is followed by profiling to search for a better equivalent version. The performance improvement over highly efficient libraries is obtained through iterative specialization, however it varies depending upon the optimizations performed by the compiler for that architecture.

7. REFERENCES

[1] M. Ahmad, H.-P. Charles, and D. Barthou. Reducing code size explosion through low-overhead specialization. In *11th Annual Workshop INTERACT-11*, Phoenix, 2007.

[2] M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski et al. Oceans: Optimizing compilers for embedded applications. In *Proceeding of Euro-Par98*, 1998.

[3] F. Bodin, T. Kisuk, P. M. W. Knijnenburg et al. Iterative compilation in a non-linear optimisation space. In *Workshop on Prole 14 and Feedback-Directed Compilation*, France, 1998.

[4] C. Consel, L. Hornof, R. Marlet et al. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998.

[5] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Intl. Conf. on Acoustics, Speech, and Signal Processing*, WA, 1998.

[6] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc : An expressive annotation-directed dynamic compiler for c. Tech. report, Univ. of Washington, 1999.

[7] http://ffmpeg.sourceforge.net. Ffmpeg, Apr. 2006.

[8] M. Leone and R. K. Dybvig. Dynamo : A staged compiler architecture for dynamic program optimization. Technical report, Indiana Univ., 1997.

[9] M. Ahmad and H.P. Charles. Improving multimedia applications through specialization of DCT/IDCT kernels. *IEEE Intl. conf. on Signal Processing and Communications(ICSPC07)*, 2007.

[10] M. Poletto, W. C. Hsieh, D. R. Engler, and F. M. Kaashoek. 'c and tcc : A language and compiler for dynamic code generation. *ACM TOPLAS*, 1998.

[11] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *27th Intl. Symposium on Microarchitecture*, 1994.

[12] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization space exploration. In *'CGO-03*, March 2003.

[13] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, Univ. of Tennessee, December 1997.

[14] H. Zhou. Code size efficiency in global scheduling for ilp processors, 2002.