An Effective Automated Approach to Specialization of Code

Minhaj Ahmad Khan, H.-P. Charles, and D. Barthou

University of Versailles-Saint-Quentin-en-Yvelines, France

Abstract. Application performance is heavily dependent on the compiler optimizations. Modern compilers rely largely on the information made available to them at the time of compilation. In this regard, specializing the code according to input values is an effective way to communicate necessary information to the compiler.

However, the static specialization suffers from possible code explosion and dynamic specialization requires runtime compilation activities that may degrade the overall performance of the application.

This article proposes an automated approach for specializing code that is able to address both the problems of code size increase and the overhead of runtime activities. We first obtain optimized code through specialization performed at static compile time and then generate a template that can work for a large set of values through runtime specialization.

Our experiments show significant improvement for different `SPEC` benchmarks on Itanium-II(IA-64) and Pentium-IV processors using *icc* and *gcc* compilers.

1 Introduction

The classical static compilation chain is yet unable to reach the peak performance proposed by modern architectures like Itanium. The main reason comes from the fact that an increasing part of the performance is driven by dynamic information which is only available during execution of the application. To obtain better code quality, a modern compiler first takes into account input data sets, and then optimizes code according to this information.

Static specialization of integer parameters provides to the compiler the opportunity to optimize code accordingly, but it comes at the expense of large code size. A wide range of optimizations can take advantage of this kind of values: branch prediction, accurate prefetch distances (short loops do not have the same prefetch distance as loops with large iteration count), constant propagation, dead-code elimination, and complex optimizations including loop unrolling and software pipelining etc. can then be performed by the compiler.

The dynamic behavior of the applications and unavailability of information at static compile time impact the (static) compilation sequence and result in specialization of code to be performed at runtime. The code is specialized and optimized during execution of the program. It is mostly achieved by dynamic code generation systems [1,2,3,4,5] and offline partial evaluators [6,7]. These

V. Adve, M.J. Garzarán, and P. Petersen (Eds.): LCPC 2007, LNCS 5234, pp. 308–322, 2008.
© Springer-Verlag Berlin Heidelberg 2008

```
void smvp(int nodes, params..) {
 for (i = 0; i < nodes; i++) {
  Anext = Aindex[i];
  Alast = Aindex[i + 1];
  sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] +
  A[Anext][0][2]*v[i][2];
  sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] +
  A[Anext][1][2]*v[i][2];
  sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] +
  A[Anext][2][2]*v[i][2];
  Anext++;
   ...
 }//end for
}//end function
```

Fig. 1. `183.equake` benchmark

systems perform runtime activities including analysis and/or computations for code generation and optimizations. All these activities incur a large overhead which may require hundreds of calls to be amortized.

For the hybrid specialization approach proposed in this paper, we do not require such time-consuming activities. The runtime specialization is performed for a limited number of instructions in a generic binary template. This template is generated during static compilation and is highly optimized since we expose some of the unknown values in the source code to the compiler. This step is similar to static specialization. The template is then adapted to new values during execution thereby avoiding code explosion as in other existing specializers. This step is similar to dynamic specialization with a very small runtime overhead. We have applied our method to different benchmarks from SPEC CPU2000 [8] suite.

The remainder of the paper is organized as follows. Section 2 describes the main principle on which hybrid specialization is based. Section 3 provides the required context that is essential to apply this technique and Section 4 elaborates the main steps included in the algorithm. The implementation details describing the input and output of each phase are provided in Section 5. Sections 6 and 7 present respectively the experimental results including the overhead incurred. A comparison with other technologies has been given in Section 8 before concluding in Section 9.

2 Principle of Hybrid Specialization

Consider the code in Figure 1 of the most time-consuming function *smvp* from `183.equake` benchmark. Code specialization is oriented towards improving the performance by simplification of computations (through constant propagation, dead code elimination), or by triggering other complex optimizations such as software pipelining. It however may result in code explosion if performed at

static compile time. Although the runtime optimizations may take advantage of known input, the cost of these optimizations makes them inappropriate to be performed at runtime.

If the parameter `nodes` is specialized with constants from 1 to 8192, we obtain different versions of code. We categorize them into different *classes* where each class differs from others in terms of optimizations but contains versions which are similar in optimizations. The versions in a class must differ only by some immediate constants. These differences occur due to different values of a specialized parameter.

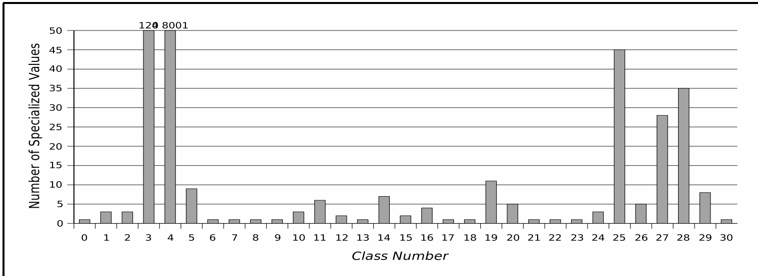


Fig. 2. *Classes* obtained for *183.equake* benchmark

Analyzing object code generated through Intel compiler *icc V9*, we find only 31 *classes* of code. Figure 2 shows the classes obtained after specialization together with the number of versions in each class. Any version in the class can serve as a *template* which can be instantiated at runtime for many values. Such behavior of compilers is similar for other benchmarks as well, even for different architectures.

The principle of the optimization we propose relies on the fact that while versioning functions for different parameter values, the compiler does not generate completely different codes. For some parameter value range, these codes have the same instructions and only differ by some constants. The value range to consider can be defined by several approaches: profiling, user-input, or static analysis. The idea is to build a binary template, which if instantiated with the parameter values, is equivalent to the versioned code. If the template can be computed at compile time, the instantiation can be performed at run-time with little overhead. We therefore have the best of versioning and dynamic specialization, i.e., we take advantage of complex static compiler optimizations and yet obtain the performance of versioned code without paying the cost of code expansion.

The hybrid specialization approach is depicted in Figure 3. The first step consists of versioning a function for appropriate parameters. From these versions, a template is extracted if this is possible. The template generation also includes the generation of a dynamic specializer together with specialized data for the template. The final hybrid code therefore comprises template versions, dynamic specializer and the original compiled code (as a fallback).

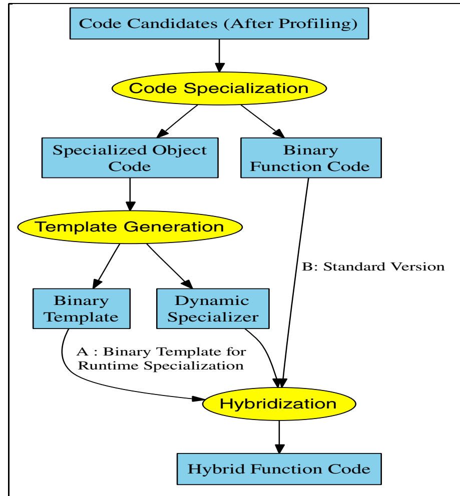


Fig. 3. Overview of Hybrid Specialization

3 Template Creation and Efficient Runtime Specialization

We define the notion of template as an abstraction of binary code with some slots (locations in the binary code) corresponding to parameters specialized. These slots can be filled with the constant values instantiating the template. Let $T_{X_1 \ldots X_n}$ denote a binary template with slots $X_1, \ldots, X_n$. The instantiation of this template with the constant integer values $v_1, \ldots, v_n$ is written $T_{X_1 \ldots X_n}[X_1/v_1, \ldots, X_n/v_n]$, and corresponds to a binary code where all slots in the template have been filled with values. The complexity of instantiation of a template is $O(n)$ which is very low as compared to full code generation and optimizations performed at runtime.

3.1 Equivalence of Specialized Binaries

Now consider the code of a function F to be optimized, we assume without loss of generality that F takes only one integer parameter X. This function is compiled into a binary function $C(F)(X)$, where C denotes the optimization sequence and code generation performed by the compiler. By versioning F with a value v, the compiler generates a binary $B_v = C(F_v)$, performing better than $C(F)(v)$. We define an equivalence between specialized binaries:

Definition 1. *Given two specialized binaries B_v and $B_{v'}$, B_v is equivalent to $B_{v'}$ if there exists a template $T_{X_1 \ldots X_n}$ and functions $f_1 \ldots f_n$ such that*

$$T_{X_1 \ldots X_n}[X_1/f_1(v), \ldots, X_n/f_n(v)] = B_v,\ T_{X_1 \ldots X_n}[X_1/f_1(v'), \ldots, X_n/f_n(v')] = B_{v'}.$$

In other words, the two specialized binaries are equivalent if they are instantiations of the same template with the same function applied to their specialized value. Let R denote this equivalence.

This is indeed an equivalence: reflexivity and symmetry are obvious, and for the transitivity: Assume $B_v \mathcal{R} B_{v'}$ and $B_{v'} \mathcal{R} B_{v''}$, for $v \neq v''$, this means that there exist two templates $T_{X_1 \ldots X_n}$ and $T_{Y_1 \ldots Y_m}$, and two sets of functions $f_1 \ldots f_n$ and $g_1 \ldots g_m$ such that:

$$\begin{array}{r}
T_{X_1 \ldots X_n}[X_1/f_1(v), \ldots, X_n/f_n(v)] = B_v \\
T_{X_1 \ldots X_n}[X_1/f_1(v'), \ldots, X_n/f_n(v')] = B_{v'} \\
T_{Y_1 \ldots Y_m}[Y_1/g_1(v'), \ldots, X_m/g_m(v')] = B_{v'} \\
T_{Y_1 \ldots Y_m}[Y_1/g_1(v''), \ldots, X_m/g_m(v'')] = B_{v''}
\end{array}$$

Assume, without loss of generality, that the first p slots $Y_1, \ldots, Y_p$ correspond to the slots $X_1, \ldots, X_p$. For these slots, we deduce from the preceding equations that $f_i(v') = g_i(v')$, for all $i \in [1..p]$. We define $m-p+1$ new functions on v' and v'' by $f_{i-p-1+n}(v') = g_i(v')$, $f_{i-p-1+n}(v'') = g_i(v'')$ for $i \in [p+1..m]$. Finally, we define these functions for v as the value in the binary B_v taken in the slot Y_i, $i \in [p+1..m]$. To conclude, we have defined a new template $T_{X_1 \ldots X_n Y_{m-p-1} \ldots Y_m}$ such that the instantiation of this template with the functions f_i in v, v' and v'' gives the binaries $B_v, B_{v'}$ and $B_{v''}$. These three binaries are equivalent, and R is an equivalence relation.

Computing the minimum number of templates necessary to account for all specialized binaries B_v when $v \in [lb, ub]$ boils down to computing the equivalence classes $\{B_v, v \in [lb, ub]\}/\mathcal{R}$ incrementally. Given below is the relation between specialized binaries and templates:

Interval of values		**Specialized binaries**		**Binary templates**
$[lb, ub]$	$\longrightarrow$	$\{B_v, v \in [lb, ub]\}$	$\rightleftharpoons$	$\{B_v, v \in [lb, ub]\}/\mathcal{R}$

As shown in the motivating example, there are many more specialized binaries than binary templates. Given the range of values to specialize for, compilation of the specialized binaries from the original code is achieved by a static compiler. Computation of the templates is likewise at static compile time. Instantiation of the templates then corresponds to the efficient dynamic specialization, performed at run-time.

3.2 Minimizing Overhead of Template Specialization

The overhead of template specialization is reduced through the generation of template at static compile time together with generation of specialized data requiring no calculation at runtime.

To compute the specialized data for instantiation of templates, we proceed after having found the *classes*. For each *class* computed from specialized binaries, let $v_t = v_1$ be first value for the *class* whose version will act as a template.

For values $v_2, v_3, \ldots, v_n$, occurring in the same equivalence *class* (producing n versions in the class),

– Initialize a linear data list with immediate values which exist in version specialized with v_t and do not exist in version specialized with v_2.
– Insert into the data list the values that differ (at corresponding locations) in the version specialized with v_t and those in versions specialized with $v_2, v_3, \ldots, v_n$.
– Generate the formula corresponding to the starting index of the element for the *class* in the data list.

By using this specialized data, it is easier to instantiate the template without calculating the runtime values.

4 Optimization Algorithm

We describe in this section the main steps that are required to perform hybrid specialization, incorporating both static and dynamic specializations. After obtaining intervals of values of the parameters, the following steps are performed.

1. Code specialization and analysis of object code;
   Different specialized versions of the function may be generated where its integer parameters are replaced by constant values. The specialized object code is analyzed to obtain a template that can be used for a large range of values. This search is performed within profiled values to meet the conditions described in Section 3. The equivalent specialized code versions differ in immediate constants being used as operands in object code. The instructions which differ in these versions will be termed as *candidate* instructions.
2. Generating statically specialized data list;
   The runtime specialization overhead is minimum if necessary data required for specialization of binary code has already been computed at static compile time. This specialized data (to be inserted into binary instructions) can be obtained for values in the interval corresponding to each *candidate* instruction as given in Section 3.2. The specialized data approach not only transfers the complexity of runtime computations to static compile time but also reduces the overhead of possible code size increase.
3. Generation of runtime specializer and template;
   For the *classes* containing more than one value, a runtime specializer is generated. The runtime specializer contains the code to search for the proper template and subsequently modify binary instructions of that template. Information regarding locations of each *candidate* instruction can be easily gathered from object code. The template in hybrid specialization therefore comprises all the *candidate* instructions to be modified during execution. The modification of instructions can then be accomplished by self-modifying code.

   This approach ensures that the cost of runtime code generation/modification is far less than that in existing specializers and code generators. The optimizations on the template have already been performed at static compile time due to specialization of code.

5 Implementation Framework and Experimentation

The hybrid specialization approach (depicted in Figure 3) has been automated for function parameters of integral data types in *HySpec*[5] framework. It takes input configuration file containing the functions, parameters, the intervals and compilation parameters. The intervals can be specified based on application-knowledge, otherwise code is first instrumented at *routine* level with *HySpec* to obtain the value profile [9] for integral parameters of the functions.

In addition to instrumentation for value profiling, *HySpec* performs different steps to generate hybridly specialized code which are given below.

5.1 Code Specialization and Object Code Analysis

Within interval values, code is specialized by exposing the values of function parameters. The code is parsed[1] to generate another specialized version. This is followed by an analysis of object code to search for *classes* of code, so that within a *class* the versions differ only in immediate constants. For example, for

Table 1. Object code generated over Itanium-II and Pentium-IV

Value	**IA-64**	**P-IV**
nodes=19	cmp.ge.unc p6,p0=**19**,r54	cmpl $**19**, %eax
nodes=17	cmp.ge.unc p6,p0=**17**,r54	cmpl $**17**, %eax

`183.equake`, the object code generated by `icc` compiler, when specialized with the value *nodes*=17 and the one generated for *nodes*=19 differs only in some constants as shown in Table 1. These instructions correspond to the value of specialized parameter.

5.2 Generation of Specialized Data and Runtime Specializer

Automatic generation of specialized data and the runtime specializer renders hybrid specialization to be a declarative approach. For an interval, all the values corresponding to each instruction differing in equivalent versions are used to generate a linear array of specialized data. This array represents the actual values with which the binary code is specialized during execution. The offset of data from where the values start for an instance of a template, are also computed at static compile time.

The template can be specialized by modifying instructions at runtime. This is accomplished by the runtime specializer which is able to efficiently insert values at specified locations. These locations are also calculated during analysis of object code. As shown in Figure 4 (on the right), each invocation of *Instruction Specializer* puts statically specialized data into template slots. This is followed by activities for cache coherence (required for IA-64).

[1] Only the C language is supported.

The *Instruction Specializer* is implemented as a set of macros which may have different functionality for different processors due to different instruction set architecture. For Itanium-II, the offset contains the bundle number and instruction number within that bundle, whereas for Pentium-IV, it contains exact offset of the instruction to be modified.

5.3 Final Wrapper Code

Figure 4 (on the left) shows the pseudo-code for the wrapper. It first searches for the template for which the new (runtime) value is valid. The branches in the wrapper are used to redirect control to the proper version. For each template, the current implementation supports dynamic specialization with a software cache of single version. We intend to implement the software cache with multiple clones to mitigate the problem of parameters with repeated patterns.

```
static long old_Param[]={...};
void WrapperFunction (Parameters)
   Let TN = FoundTemplate
   if TN>0 then
     if Param <> old_Param[TN]
       Branch to Specializer[TN]
       Update old_Param
     end if
     Branch to Template[TN]
   else
     Branch to Standard code
End Function
```

```
Offset = Location of candidate inst.
Data = Pointer to specialized data
BA = Function's Base Address
void BinaryTemplateSpecializer{
  InstSpec(BA+offset_0, Data[0])
  InstSpec(BA+offset_1, Data[1])
  InstSpec(BA+offset_2, Data[2])
  InstSpec(BA+offset_3, Data[3])
  .........
  .........
  .........
}
```

Fig. 4. Wrapper code (left) and invocation of Instruction Specializer(right)

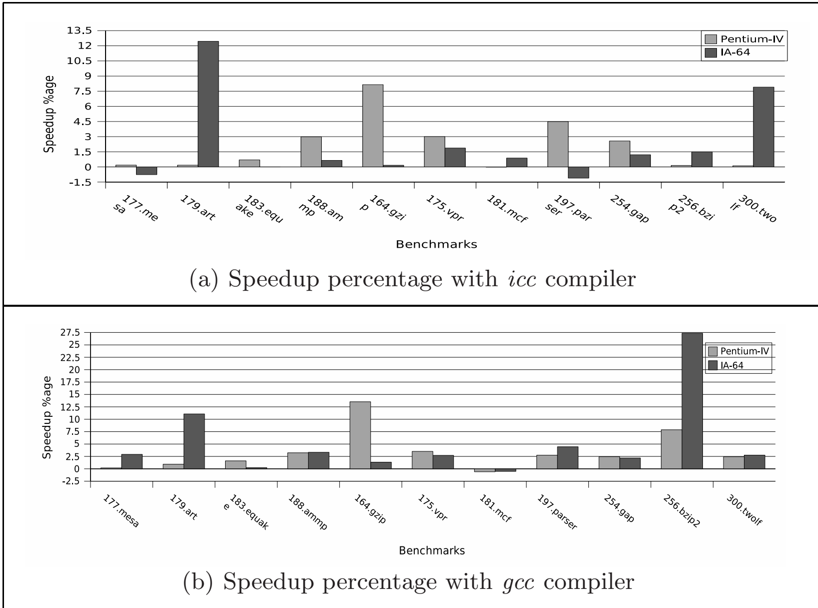
6 Experimental Results

The specialization approach has been applied to hot functions in SPEC CPU2000 benchmarks with reference inputs. The experiments have been performed over platforms with the configurations given in Table 2. This section describes the results of these benchmarks together with optimizations performed by compilers due to specialization.

Figure 5 shows the speedup percentage obtained w.r.t standard (original) code. For these benchmarks, the speedup largely depends upon the use of parameters in the code. The hot code of these benchmarks does not always include

Table 2. Configuration of the Architectures Used

Processor	Speed	Compilers & optimization level
Intel Itanium-II (IA64)	1.5 GHz	gcc v 4.3, icc v 9.1 with -O3
Intel Pentium-4 (R)	3.20 GHz	gcc v 4.3 , icc v 8.0 with -O3



(a) Speedup percentage with *icc* compiler

(b) Speedup percentage with *gcc* compiler

Fig. 5. Performance Results of `SPEC CPU2000` Benchmarks

integer parameters, and in some cases, the candidate parameters were unable to impact overall execution to a large factor.

For benchmark *mesa*, the compilers were able to perform inlining and partial evaluation. However, these optimizations did not put any significant impact on execution time. In *art* benchmark, the main optimizations were data prefetching and unrolling which resulted in good performance on IA-64 architecture. However, the compilers did not make any big difference w.r.t standard code on Pentium-IV architecture.

For *equake*, the large values of specializing parameters resulted in code almost similar to that of un-specialized version with small difference in code scheduling. Similarly, the *gzip* benchmark benefits from loop optimizations and code inlining on Pentium-IV, however on Itanium-II, the compilers generated code with similar unroll factor for both the standard and specialized versions.

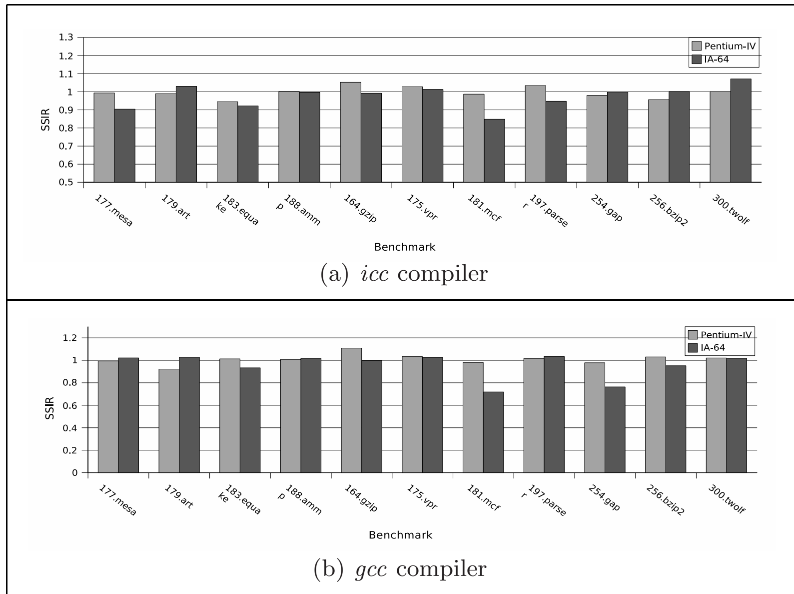
In the *ammp*, *vpr*, *mcf*, *parser* and *gap* benchmarks, a large part of hot functions does not make use of integer parameters and the large frequency of variance in runtime values reduces the performance gain after hybrid specialization.

In case of the *bzip2* benchmark, the *gcc* compiler performed partial evaluation and the loop-based optimizations which did not exist in un-specialized code. With the *icc* compiler, the loop-based optimizations were similar in both the specialized and un-specialized code with small difference due to partial evaluation.

The *twolf* benchmark benefits mainly from data cache prefetching, reduced number of loads and better code scheduling on IA-64 with *icc* compiler, whereas

Table 3. Summarized analysis for SPEC benchmarks

Benchmark	**Number of static versions reqd.**	**Percentage of re-instantiations**				**Number of Templates**				**Percentage of code size increase (w.r.t. un-specialized benchmark)**			
		IA-64		P-IV		IA-64		P-IV		IA-64		P-IV	
		icc	gcc	icc	gcc	icc	gcc	icc	gcc	icc	gcc	icc	gcc
177.mesa	9	8%	8%	36%	8 %	9	9	2	9	10%	1%	1%	1%
179.art	5	1%	1%	1%	1%	4	5	5	4	9%	8%	9%	1 %
183.equake	1	0%	0%	0%	0%	1	1	1	1	8%	7%	1%	7%
188.ammp	1	0%	0%	0%	0%	1	1	1	1	1%	2%	1%	3%
164.gzip	15	43%	86%	43%	86%	2	1	2	1	1%	2%	1%	3%
175.vpr	8444	1%	1%	1%	1%	3	3	11	4	1%	1%	1%	1%
181.mcf	10	1%	1%	1%	1%	3	1	3	1	19 %	38 %	19%	1%
197.parser	40	21%	42%	21%	42%	2	1	2	1	4%	1%	4%	1%
254.gap	53	8%	16%	8%	63%	8	4	8	1	34%	10%	1%	5%
256.bzip2	5	21%	42%	21%	21%	2	1	2	2	2%	1%	1%	5 %
300.twolf	2	49%	49%	49%	49%	1	1	1	1	1%	1%	1%	1%



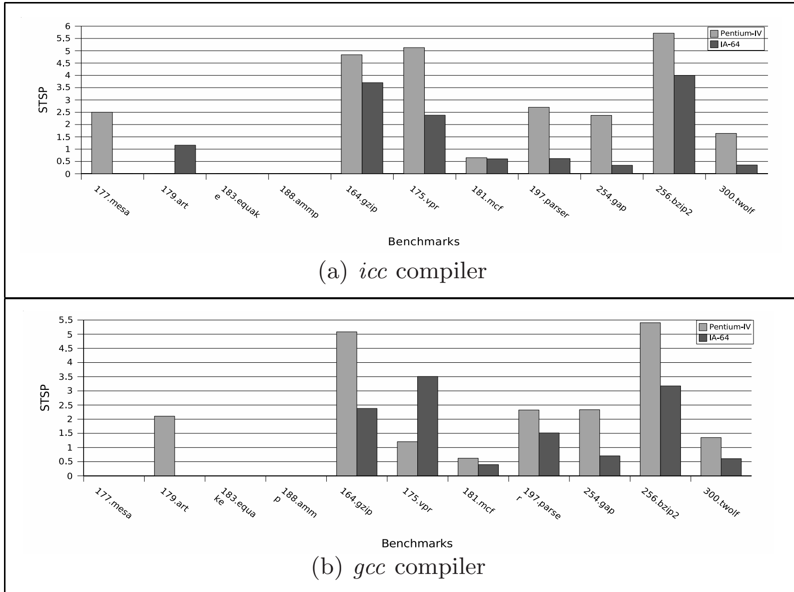
(a) *icc* compiler

(b) *gcc* compiler

Fig. 6. Speedup to Size Increase Ratio(SSIR) for SPEC Benchmarks. SSIR=1 means that the speedup obtained is equal to the code size expansion.

for the remaining platform configurations, the compilers were limited to performing inlining and partial evaluation.

Table 3 shows (in column 1) the number of versions that were required for static specialization together with percentage of re-instantiations of the same



(a) *icc* compiler

(b) *gcc* compiler

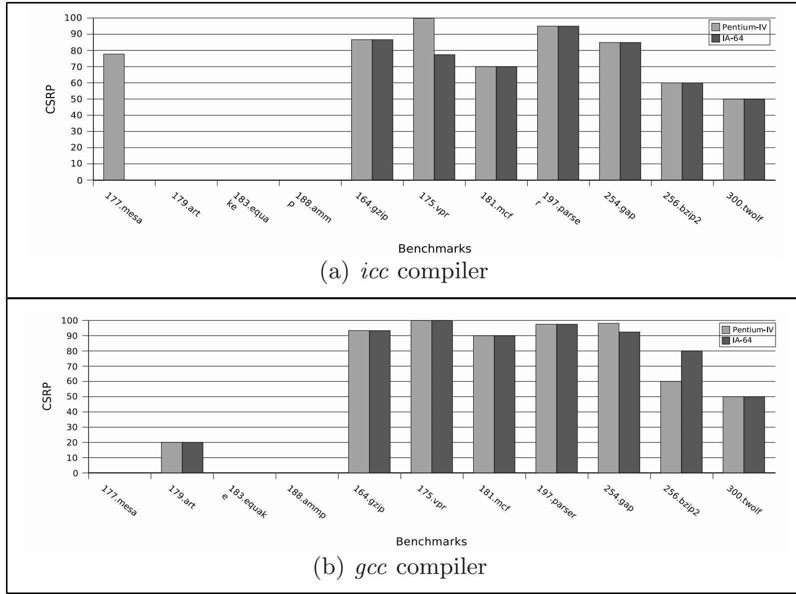
Fig. 7. Slot to Template Size Percentage (STSP) for `SPEC` Benchmarks

template (in column 2). The large percentage of re-instantiations for the *gzip*, *parser* and *twolf* benchmarks represents repeated pattern of values. This factor can only be minimized through software cache of templates which is part of future work.

Columns 3 and 4 show respectively the number of templates and the percentage of code size increase w.r.t. un-specialized code. The compilers show variant behaviour in terms of code size after code specialization mainly due to different optimizations. This is why, sometimes code with a large number of specialized versions/templates may result in less size than with a small number of specialized versions.

The speedup to size increase ratio (SSIR) computed as $\frac{Speedup}{\left(\frac{Size\ of\ code\ after\ specialization}{Size\ of\ unspecialized\ code}\right)}$ for `SPEC` benchmarks has been given in Figure 6. The SSIR metric is a measure of efficiency for our specialization approach (similar to the one used for parallelism). The SSIR is not large over both the processors even with benchmarks having large speedup, e.g., *art*. This is due to the fact that for benchmarks with the large speedup, standard (un-specialized) code size of entire application is small, and addition of hybrid code with specialized versions, template and specializer code thereby reduces the SSIR factor.

The Figure 7 shows the largest slot to template size percentage (STSP) for each benchmark. It is calculated as: $\left(\frac{No.\ of\ slots\ reqd.\ for\ dynamic\ specialization}{Total\ no.\ of\ instructions\ in\ template}\right) * 100$. For benchmarks *mesa*, *equake*, *ammp*, where the number of static versions required is equal to the number of templates, it becomes zero. However, it is less than 6%



(a) *icc* compiler

(b) *gcc* compiler

Fig. 8. Code Size Reduction Percentage(CSRP) for SPEC Benchmarks w.r.t equivalent static specialization. 50% would mean that the number of templates is half the number of static versions required to cover the same specialized values.

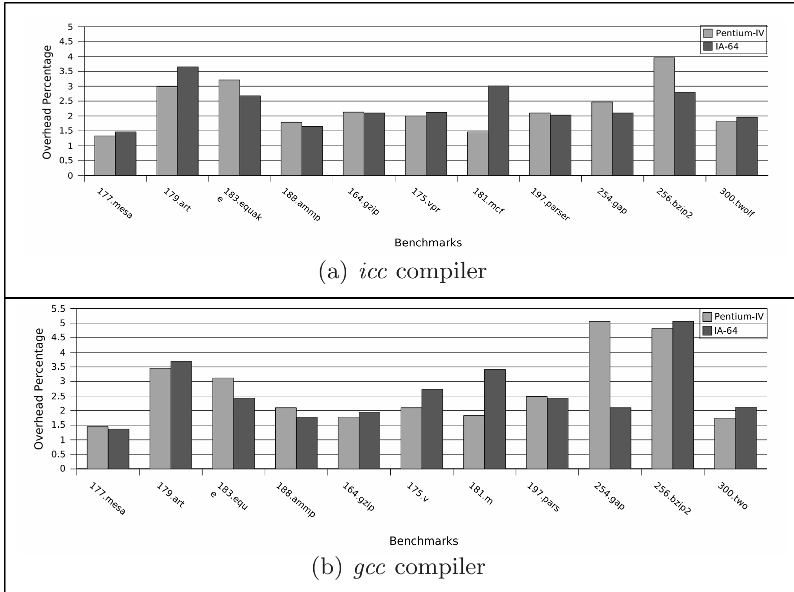
for all benchmarks which shows that our specialization method incurs the smallest possible overhead at runtime.

The effectiveness of hybrid specialization also lies in code size reduction w.r.t static specialized code for the same input intervals. In this regard, the metric *Code Size Reduction Percentage* calculated as, $\left(1 - \frac{\textit{Number of Templates found}}{\textit{Number of Static Versions Required}}\right) * 100$, has been given in Figure 8. For the benchmarks *mesa*, *art*, *equake* and *ammp*, the $CSRP$ is very small since the number of templates is very close to the number of versions required for static specialization. For other benchmarks, this factor becomes large since a single template is used to serve a very large number of values.

7 Specialization Overhead

A summarized view of overhead with respect to application execution time is shown in Figure 9. The reduced overhead results in good performance for SPEC benchmarks. It is due to the fact that the templates and the values to be inserted at runtime are entirely generated at static compile time. The modification of a single instruction takes an average[2] of 9 cycles on Itanium-II and 2 cycles on Pentium-IV. This overhead of generation of instructions is far less than that in existing dynamic compilation/specialization systems e.g. Tempo [6] or Tick C [1],

[2] The binary instruction formats require extraction of different numbers of bit-sets.



(a) *icc* compiler

(b) *gcc* compiler

Fig. 9. Overhead of Specialization w.r.t Execution Time

where it takes 100 cycles with the VCODE interface (with no optimizations) and 300 to 800 cycles using the ICODE interface (with optimizations) to generate a single instruction.

Moreover, the time taken to generate templates at static compilation depends upon the size of intervals. For benchmark with the largest interval size, i.e. *vpr*, it takes 5 hours for *gcc* on IA-64, otherwise it takes 3 hours for all other configurations.

8 Related Work

The C-Mix [10] partial evaluator works only at static compile time. It analyzes the code and makes use of specialized constructs to perform partial evaluation. Although it does not require runtime activities, it is limited to optimizing code for which the values already exist. The scope therefore becomes limited since a large part of application execution is based on values only available during execution.

The Tempo [6] specializer can perform specialization at both static compile time and runtime. At static compile time, Tempo performs partial evaluation that is only applicable when the values are static (i.e. already known). In contrast, hybrid specialization makes the unknown values available and uses a template that is already specialized at static compile time. Therefore, the template is more optimized in our case than the one generated through the Tempo specializer. Similarly, another template-based approach of specialization has been

given in [4,5]. They suggest the use of affine functions to perform runtime code specialization. The scope of templates therefore becomes very limited since the number of constraints for generating templates is very large. Moreover, the runtime computations required for specialization of templates reduce the impact of optimizations.

The Tick C(‘C)[1] compiler makes use of the *lcc* retargetable intermediate representation to generate dynamic code. It provides ICODE and VCODE interfaces to select the trade-off between performance and runtime optimization overhead. A large speedup is obtained after optimizations during execution of code. However, its code generation activity incurs overhead that requires more than 100 calls to amortize. In case of the hybrid specialization approach, we minimize the runtime overhead through generation of optimized templates and specialized data at static compile time. Similarly, most of the dynamic code generation and optimization systems like Tick C [1], DCG [11] or others suggested in [2,12,7] are different in that these can not be used to produce *generic* templates thus requiring large number of dynamic template versions for each different specializing value. The runtime activities other than optimizations, such as code buffer allocation and copy, incur a large amount of overhead thereby making them suitable for code to be called multiple times.

In runtime optimization systems, Dynamo [13] and ADORE [14] perform optimizations and achieve good speedup, but these systems do not provide the solution to control code size increase caused by dynamic versions.

9 Conclusion and Future Work

This article presents a hybrid specialization approach which makes use of static specialization to generate templates that can be specialized at runtime to adapt them to different runtime values. For many complex `SPEC` benchmarks, we are able to achieve good speedup with minimum increase in code size. Most of the heavy-weight activities are performed at static compile time including the optimizations performed by compilers. The code is specialized statically and object code is analyzed to search for templates followed by generation of a runtime specializer. The specializer can perform runtime activities at the minimum possible cost.

A generalization mechanism makes the template valid for a large number of values. This new concept of template serves two purposes: to control the code size with minimum runtime activities and benefit from optimizations through specialization performed at static compile time.

The current implementation framework of hybrid specialization is being embedded into XLanguage [15] compiler with additional support of software cache containing more clones of same templates.

References

1. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, F.M.: ’C and tcc: A language and compiler for dynamic code generation. ACM Transactions on Programming Languages and Systems 21, 324–369 (1999)

2. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: An expressive annotation-directed dynamic compiler for C. Technical report, Department of Computer Science and Engineering, University of Washington (1999)
3. Leone, M., Lee, P.: Optimizing ml with run-time code generation. Technical report, School of Computer Science, Carnegie Mellon University (1995)
4. Khan, M.A., Charles, H.P.: Applying code specialization to FFT libraries for integral parameters. In: 19th Intl. Workshop on Languages and Compilers for Parallel Computing, New Orleans, Louisiana, November 2-4 (2006)
5. Khan, M.A., Charles, H.P., Barthou, D.: Reducing code size explosion through low-overhead specialization. In: Proceeding of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture, Phoenix (2007)
6. Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.N.: Tempo: Specializing Systems Applications and Beyond. ACM Computing Surveys 30(3es) (1998)
7. Consel, C., Hornof, L., Noël, F., Noyé, J., Volanschi, N.: A uniform approach for compile-time and run-time specialization. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Partial Evaluation, Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 54–72. Springer, Heidelberg (1996)
8. SPEC: SPEC Benhmarks: SPEC (2000), http://www.spec.org/cpu2000/
9. Calder, B., Feller, P., Eustace, A.: Value profiling. In: International Symposium on Microarchitecture, pp. 259–269 (1997)
10. Makholm, H.: Specializing C— An introduction to the principles behind C-Mix. Technical report, Computer Science Department, University of Copenhagen (1999)
11. Engler, D.R., Proebsting, T.A.: DCG: An efficient, retargetable dynamic code generation system. In: Proceedings of Sixth International Conf. on Architectural Support for Programming Languages and Operating Systems, California (1994)
12. Leone, M., Lee, P.: Dynamic Specialization in the Fabius System. ACM Computing Surveys 30(3es) (1998)
13. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. ACM SIGPLAN Notices 35(5), 1–12 (2000)
14. Lu, J., Chen, H., Yew, P.C., Hsu, W.C.: Design and Implementation of a Lightweight Dynamic Optimization System. Journal of Instruction-Level Parallelism 6 (2004)
15. Donadio, S., Brodman, J., Roeder, T., Yotov, K., Barthou, D., Cohen, A., Garzaran, M., Padua, D., Pingali, K.: A language for the comParallel Architectures and Compilation Techniques representation of multiple program versions. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339. Springer, Heidelberg (2006)