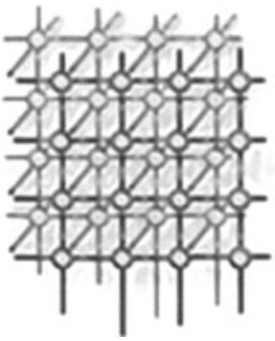
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:59–70
Published online 22 July 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1333



Improving performance of optimized kernels through fast instantiations of templates

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

University of Versailles, Versailles, France

SUMMARY

To fully exploit the instruction-level parallelism offered by modern processors, compilers need the necessary information available during the execution of the program. This advocates for iterative or dynamic compilation. Unfortunately, dynamic compilation is suitable only for applications where the cost of compilation may be amortized by multiple invocations of the same code. Similarly, the cost of iterative compilation makes it impractical to be widely used for performance improvement. In this article, we suggest a novel approach for improving the performance of mathematical kernels through fast instantiations of templates. Optimized templates are generated at static compile time with a limited number of compilations. The initial instantiations of these templates are performed at static compile time, and the runtime instantiations are performed with a very small overhead through specialized data, requiring no computations at runtime. It represents an effective solution in terms of reduced overhead incurring at static compile time and dynamic compile time. The experiments have been performed on an Itanium-II architecture using highly optimized kernels of `ATLAS` and `FFTW` with *icc* and *gcc* compilers. Copyright © 2008 John Wiley & Sons, Ltd.

KEY WORDS: code specialization; optimizations; compilation techniques

1. INTRODUCTION

Code specialization [1,2] has proved to be beneficial for many applications. Given specialized values, the compiler is able to generate highly optimized code. This allows the compiler to fully exploit the ILP provided by modern processors. Code specialization is more effectively performed at runtime due to the unavailability of values at static compile time. Runtime information may be used by many optimizations: it may remove dependencies by providing information related to array indexes or more aggressive optimizations can be invoked if the compiler is given loop

*Correspondence to: Minhaj Ahmad Khan, University of Versailles, Versailles, France.
†E-mail: Minhaj.Khan@prism.uvsq.fr

counts. Most of the optimizations such as constant propagation, dead code elimination, loop unrolling are related to integer parameters and, therefore, we target only integer parameters in this article.

A difficulty is that code specialization requires the information that is not available at static compile time. Therefore, keeping different code versions for all possible values actually degrades the performance. The dynamic compilation can mitigate this problem by generating code during execution of the application. Runtime optimization and specialization systems [2–7] may improve performance but require many invocations of the same code to amortize the overhead of runtime code generation. Similarly, the overhead of iterative compilation [8,9] has to be reduced to make it effective enough to achieve the best performance within a small time limit.

The optimization approach suggested in this article incorporates fast instantiations of the templates performed at both static compile time and dynamic compile time. An optimized template is generated after specializing the code at static compile time. This template can be used for a large set of values and requires a limited set of binary instructions to be specialized during execution. This is referred to as the instantiation of the template. While instantiating template, we also use the specialized data generated at static compile time. As the optimizations have already been performed at static compile time, the overhead of code generation is much reduced than the overhead of other dynamic code generation and optimization systems. Moreover, the static compile time overhead is reduced by using analysis and code validation against the required criteria. This optimization approach produces significant improvement in the performance of `FFTW` and `ATLAS` kernels.

The remainder of this paper is organized as follows. Section 2 discusses the compiler behavior with an example. Section 3 provides the conditions and the criteria that are essential to apply this technique. The steps included in the algorithm are elaborated in Section 4. The implementation framework has been discussed in Section 5 with experimental results presented in Section 6.

2. MOTIVATING EXAMPLE

The impact of code specialization varies depending upon the code and the use of parameter in the code. It may result in the partial evaluation with constant folding, constant propagation and dead-code elimination. Similarly, for parameters involved in the loop control (stride or bounds), it can lead the compiler to fully unroll, change the schedule of the loop (parameters of the software pipeline for instance) or perform other loop transformations (fusion for instance).

Consider the code of a BLAS-1 kernel in `ATLAS` in Figure 1 [10]. To specialize this code, one or more parameters are replaced with constants from 1 to 4096 to generate different versions. Analyzing the object code generated by the *icc* compiler V9.1 reveals that many of these versions differ only in immediate constants. The specialization produces versions that are similar for intervals of specialized values (shown in Figure 2). Any of these similar versions can be used as a template [6,9,11] to perform the functionality of other versions by changing values that differ.

This runtime adaptation of templates can be improved by performing runtime activities that do not require any computations. Moreover, the template generation must be efficient enough so as to obtain the maximum benefit within a limited time instead of acquiring an exhaustive approach. Both of these characteristics enhance the scope of template-based specialization approaches.

DOI: 10.1002/cpe

```
void ATL_UAXPY(const int N, const SCALAR alpha, const TYPE *X,
const int incX, TYPE *Y, const int incY)
{
   int i;
   for (i=0; i < N; i++, X += incX, Y += incY)
   *Y += alpha * *X;
}
```

Figure 1. ATLAS BLAS-1 KERNEL.

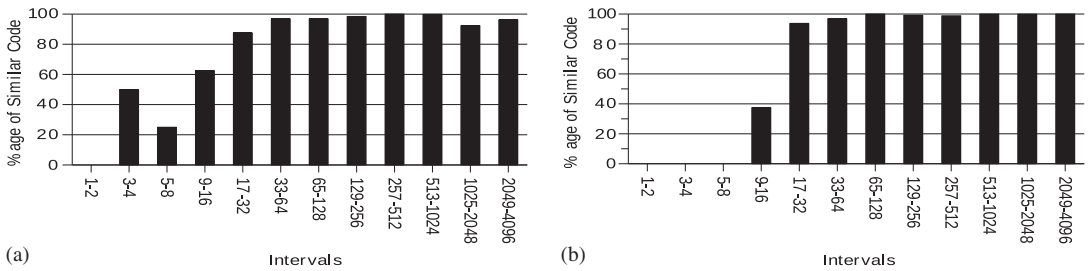


Figure 2. Code generated by the *icc* compiler (v 9.1) after specialization of code: (a) `incX` specialized and (b) `N and incX` specialized.

This article proposes an optimization approach that is based on the efficient instantiations of the templates at static compile time and dynamic compile time. This procedure involves the instantiation of the templates at static compile time for the initial value followed by the generation of a lightweight runtime specializer. The runtime specializer performs the specialization of a limited set of binary instructions to instantiate templates at runtime.

3. TEMPLATE GENERATION AND VALIDATION

This section describes how a template is generated and the required context for which it is valid.

Given an interval of parameter values, a specialized binary code can be generated by the static compiler. Let S be a function that takes as arguments the code of a function C and the value of some function parameter, and generates a specialized version. Given two values p_1 and p_2 of a parameter, we obtain two versions of a specialized binary code, $S(C, p_1) = S_{p1}$ and $S(C, p_2) = S_{p2}$, which fulfill the following criteria:

$$S_{p1} - S_{p2} = D_{p1} \tag{1}$$

$$S_{p2} - S_{p1} = D_{p2} \tag{2}$$

where D_{p1} and D_{p2} are the sets of some immediate values. This implies that the versions differ only in some constants.

DOI: 10.1002/cpe

Each $D_{p1}[i] \in D_{p1}$ and $D_{p2}[i] \in D_{p2}$ should be based on affine formulae as given in the following equations:

$$D_{p1}[i] = f_i(p_1) = p_1 * \alpha_i + \beta_i \quad \forall i = 1 \ldots n \tag{3}$$

$$D_{p2}[i] = f_i(p_2) = p_2 * \alpha_i + \beta_i \quad \forall i = 1 \ldots n \tag{4}$$

where α and β are constants and $n = |D_{p1}| = |D_{p2}|$.

Once identified, the set of such instructions is *annotated* and is used for the generation of runtime specializer.

3.1. Code validation

A runtime instantiation of the templates corresponds to the dynamic specialization performed at runtime. As this is only a substitution, the new value in the template slot (immediate operand) must be valid to be compatible with the maximum and minimum values of an instruction in the code. For runtime value $v \in V$, and the coefficients α and β corresponding to n *annotated* instructions, we must have

$$I_{\max} \geqslant v * \alpha_i + \beta_i \geqslant I_{\min} \quad \forall i = 1 \ldots n \tag{5}$$

where $I_{\max}$ and $I_{\min}$ are the maximum and minimum values that an immediate operand can have in the *annotated* instruction.

For the runtime value v, the code must also be validated against the predicates. Let $T_i(v)$ be the ith predicate involving parameter specialized with the value v, the new instantiation for v should be valid iff the following predicates are equal:

$$T_i(p_1) = T_i(p_2) = T_i(v) \tag{6}$$

The specialized code can now be instantiated as described below.

3.2. Fast template instantiations for mathematical kernels

In order to reduce the overhead at static compile time and runtime, we generate a static specialized data array. This is achieved by restricting the template values to be based only on affine functions. It means that the functions $f_1, \ldots, f_n$ required for the instantiation of a template must be of the form: $f_i(x) = \alpha_i.x + \beta_i$, where x is a parameter value with which the code is specialized.

Given an interval (of values) V, the specialized data corresponding to a runtime value $v \in V$ can be generated by solving the equations and evaluating the formula. Therefore, we have $D_v[i] = \alpha_i.v + \beta_i$. The starting index of specialized data in the D_v array (corresponding to each instantiation of the template) is also computed at static compile time.

With the specialized data, it becomes easier to instantiate the template without calculating any runtime values for the instructions.

DOI: 10.1002/cpe

4. APPROACH OF FAST TEMPLATE-BASED INSTANTIATIONS

For an interval in a value profile [12], the code is specialized by defining values (taken from the interval) for the parameters. The following steps are then required to perform fast instantiations of the optimized code obtained after specialization.

1. *Code validation*: The specialized code is analyzed to obtain a template that can be used for a large range of values. A template is searched by finding the versions that meet conditions (1)–(6) described in Section 3. The instructions differing (by immediate constants) in these versions are *annotated* to calculate the locations to be modified at runtime. Any of these versions can be used as a template, and may now be instantiated.
2. *Initial instantiations of the template with the specialized data*: The instantiations of the template require specialized data that can be generated at static compile time. For the equivalent versions specialized with $v_1, v_2, \ldots, v_k$, values of k parameters, and having n differences, we compute data through formulae of the form: $D_{v_1,v_2,\ldots,v_k}[i] = \sum_{j=1}^{k}(\alpha_{ij} \times v_j + \beta_{ij})$ for $1 \leqslant i \leqslant n$. It represents the specialized data (for each *annotated* instruction in the template) that will be inserted during execution. The template is then instantiated at static compile time by modifying its object code instructions.
3. *Runtime instantiations through the specializer*: The runtime specializer contains the self-modifying code that can insert specialized data at specified locations. The information regarding precise location of each *annotated* instruction is also calculated at static compile time. As the specialized data are already generated at static compile time, the runtime instantiations require no computations for specializing the code.

The static versions are used for interval values where the template could not be generated. A small piece of wrapper code is also generated to redirect the control to the template code, to statically specialized code and unspecialized code.

5. IMPLEMENTATION FRAMEWORK AND EXPERIMENTATION

The main steps (shown in Figure 3) leading toward fast instantiations of templates have been automated in the *HySpec* [6] framework. It supports the specialization of function parameters (integral) as described below.

5.1. Instrumentation and code specialization

The specialization of code proceeds by defining values (obtained through instrumentation for value profiling [12]) of function parameters to generate versions. The code is also instrumented to contain a call to wrapper function that can redirect the control to a specific version.

5.2. Generation of data for fast template instantiations

An analysis of object code versions is performed to validate so that the two versions differ only in immediate constants and these constants must be based on affine functions (shown in Figure 4(a)

DOI: 10.1002/cpe

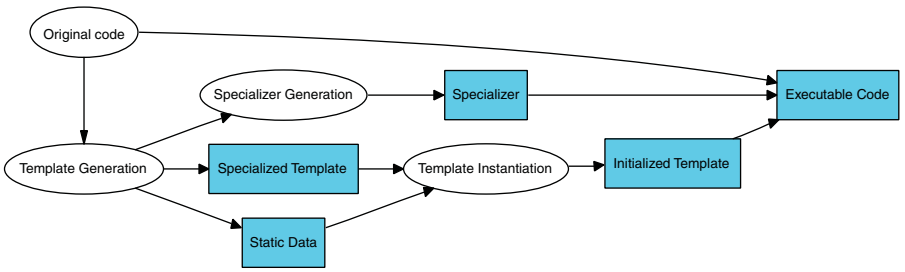


Figure 3. An overview of optimization approach.

```
add r2= 832, r34
ldfd f9= [r34], 104
lfetch.nt1 [r2], 104
```
(a)

```
add r2= 704, r34
ldfd f9= [r34], 88
lfetch.nt1 [r2], 88
```
(b)

Figure 4. Object code generated by `icc` over Itanium-II: (a) `incX` = `13` and (b) `incX` = `11`.

and (b)). After checking the code equivalence, the formulae are generated by solving the system of equations. The formulae are assumed to be of the form: $v = \alpha \times \text{specialized\_value} + \beta$. For an interval, all the values v corresponding to each instruction differing in equivalent versions are evaluated through the formula and a linear array of specialized data is generated. This results in the minimum overhead incurring at static compile time.

The specialized data array represents the values with which the binary code will be specialized. The offset of data from where the values start for an instance of template are also computed at static compile time.

5.3. Wrapper code and initial template instantiation

The call to wrapper is already instrumented, but its code is generated after the generation of templates. The branches in the wrapper redirect the control to proper versions. For some valid range of interval, it contains calls to statically specialized code, dynamically specialized template and original code as fallback. For dynamic templates, it first contains call to specializer followed by the call to template code.

The initial instantiation of the template takes place by modifying object code instructions at static compile time. The addresses of these instructions have already been computed after analysis and represent the template slots to be modified.

The runtime instantiation of the template is accomplished by a runtime specializer (shown in Figure 5), which is able to efficiently insert values at specified locations. The same locations in the object code as computed for initial instantiation are re-used for runtime specialization. Each invocation of *Instruction Specializer* puts statically specialized data into these locations followed by the cache coherence.

DOI: 10.1002/cpe

```
D15 = Array of data for specialization with value 15
   InstructionSpecializer(Instruction Address0, D15[0]) //D15[0]=960
   InstructionSpecializer(Instruction Address1, D15[1]) //D15[1]=120
   InstructionSpecializer(Instruction Address2, D15[2]) //D15[2]=120
   CacheCoherence(Instruction Address0)
    .........
```

Figure 5. Binary template specializer for a runtime value of 15.

For the template to be specialized with a runtime value of 15, the coefficients $\alpha_1 = 64$, $\alpha_2 = 8$, $\alpha_3 = 8$ and $\beta_1 = 0$, $\beta_2 = 0$, $\beta_3 = 0$ will generate 960, 120 and 120, respectively, as elements of data array.

6. EXPERIMENTAL RESULTS

The experiments for fast instantiations of the templates have been tested on IA-64 (1.5 GHz, 32 kB L1I + D, 256 kB L2 and 3 MB L3 cache) platform with the *icc* compiler v 9.1 and *gcc* compiler v 4.3. For compilation, the *-O3* optimization has been used together with default parameters with which these libraries were configured.

6.1. ATLAS Benchmarks

ATLAS [10] library generates the optimized code for the specific architecture by making use of optimizations offered by the processor and compiler with which it is configured. The BLAS-3(dgemm), BLAS-2(dgemv) and BLAS-1(daxpy) kernels in the ATLAS library have been specialized. The speedup percentage obtained with respect to the standard ATLAS code has been shown in Figures 6–8, respectively.

For BLAS-3, the code specialization has been applied to different modules[‡] *ATL_dJIK*. Similarly, for BLAS-1 and BLAS-2, the routine *ATL_daxpy_xp*1*yp*1*aXbX* has been specialized. The specialized codes (six versions) mostly benefit from loop-based optimizations including data prefetching, software pipelining and loop unrolling.

A significant improvement in the performance of the BLAS kernels is obtained with the *icc* compiler as the ATLAS standard code (obtained after tuning) contains routines that are not very much specialized. This is similar in case of *gcc* compiler, for which performance is gained for dgemv and daxpy. However, in dgemm, the code is already specialized to a large extent. This restricts the compiler to only perform partial evaluation together with a small reduction in the number of loads after code specialization.

[‡]Functions with a1_b1 and aX_bX suffixes.

DOI: 10.1002/cpe

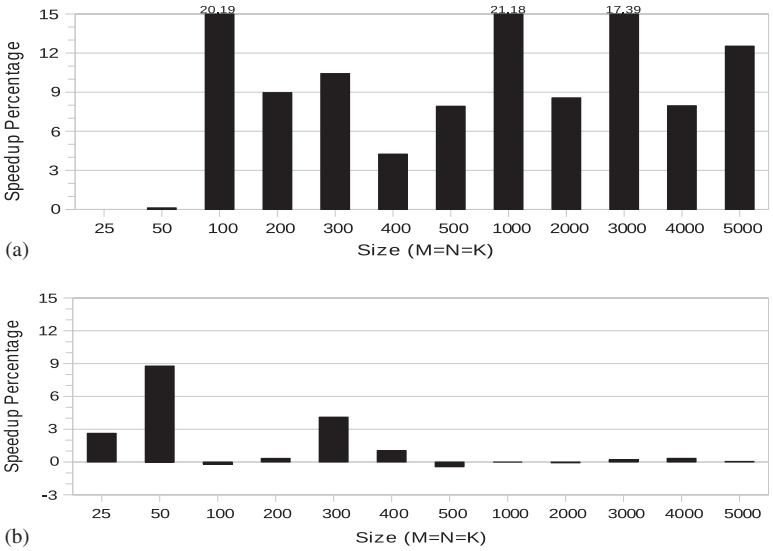


Figure 6. BLAS-3(DGEMM) speedup percentage w.r.t. standard BLAS-3 code: (a)`icc` compiler and (b) `gcc` compiler.

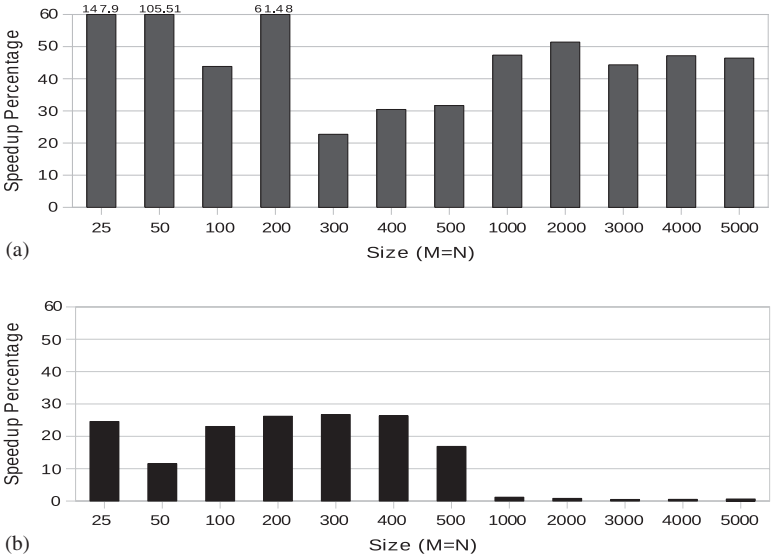


Figure 7. BLAS-2(DGEMV) speedup percentage w.r.t. standard BLAS-2 code: (a) `icc` compiler and (b) `gcc` compiler.

DOI: 10.1002/cpe

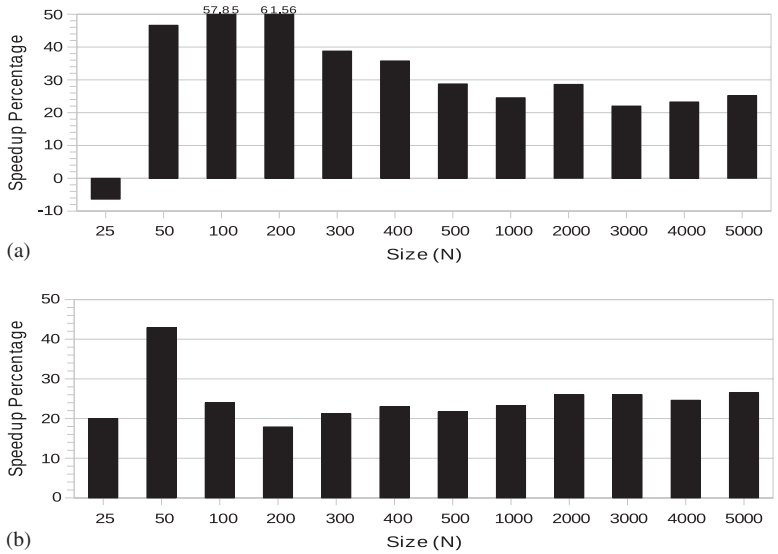


Figure 8. BLAS-1(DAXPY) speedup percentage w.r.t. standard BLAS-1 code: (a) *icc* compiler and (b) *gcc* compiler.

6.2. `FFTW` Benchmark

`FFTW` [13] library contains C routines called *codelets* to compute discrete Fourier transform (DFT) of real and complex data and of arbitrary input size in $O(n \log n)$. It makes use of the best configuration for the corresponding architecture called *wisdom*.

Figure 9 shows the speedup obtained for calculating complex DFTs of powers of 2. The calculation of single DFT may comprise repeated invocations of multiple codelets.

The overall behavior of the compilers optimization depends upon both the value of the specialized parameter and the size of the specialized code. For large codelets in FFTW, the code generated by the compiler becomes very similar to the unspecialized code thereby producing less improvement in performance.

6.3. Overhead of runtime instantiations

The overhead[§] of runtime instantiations (shown in Figure 10) is very small as the initial instantiations have already been made at static compile time. Moreover, the static specialized data eliminates the need for any runtime computation. The modification of single instruction takes an average of nine cycles on Itanium-II. The calculation of offset of specialized data and cache coherence are both performed once per runtime specialization of the entire template.

[§]Calculated through profiling of wrapper that contains specializer invocation.

DOI: 10.1002/cpe

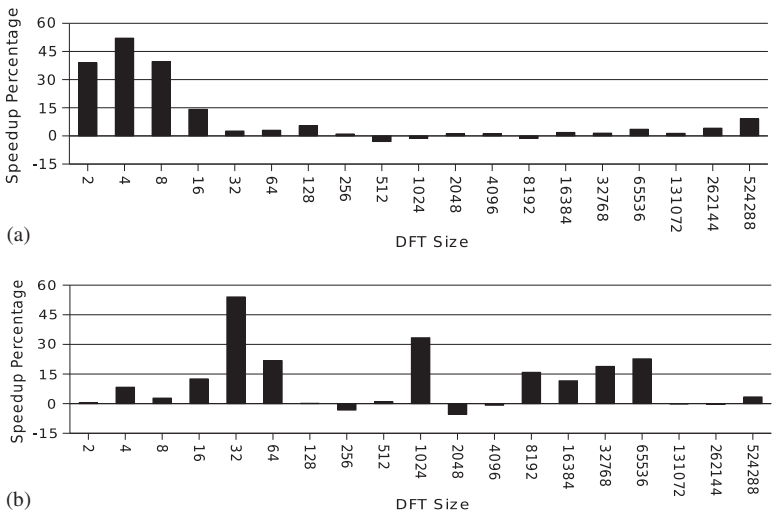


Figure 9. FFTW speedup w.r.t. standard code: (a) icc compiler and (b) gcc compiler.

Benchmark	Avg. Overhead		Avg. SI		Avg. PUI	
	icc	*gcc*	*icc*	*gcc*	*icc*	*gcc*
FFTW	3.97%	3.91%	2.31	2.67	10.5%	9.8%
BLAS	1.60%	1.77%	3.14	3.49	7.0%	5.4%

Figure 10. Summary of overhead, code size increase and percentage of unique instantiations of the templates.

The size increase (SI) which is calculated as *size of specialized code*/*size of unspecialized code*, and percentage of unique instantiations (PUI), calculated as (*number of unique runtime instantiations*/*number of calls*)∗100, are also provided in Figure 10. The SI becomes large particularly for small unspecialized code for which the addition of template size and static code becomes significant. The PUI for both the benchmarks is small, as the kernels do not frequently change values. Moreover, the initial instantiation eliminates the need for dynamic instantiation.

7. RELATED WORK

Many code specialization and dynamic code generation systems use templates for performing optimizations at runtime.

The Tempo specializer [2] performs partial evaluation of code by propagating information after analysis of code. The dynamic code can then be generated by specializing the code with runtime values. In contrast, our approach performs fast instantiations of the templates optimized at static compile time and, therefore, it incurs minimum overhead at runtime.

Our previous work on dynamic specialization [6] uses affine formulae to be computed at runtime. Although this approach incurs small overhead compared with other specializers, the computations

DOI: 10.1002/cpe

are required before generating a new instruction. In contrast, the approach suggested in this paper incorporates specialized data computed at static compile time. This eliminates the need to perform any computations at runtime. The instantiations of the templates therefore incur the smallest possible overhead. Similarly, another specialization approach described in [9] uses exhaustive specialization of code at compile time to generate the templates. In contrast, the templates for fast instantiations are generated with the minimum number of compilations. The overhead of iterative compilation is reduced by using the analysis and validation criteria that enables a large range of specialized data to be generated through computation of affine formulae.

C-Mix [14] partial evaluator performs source-to-source transformation 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 the code for which the values already exist, thereby limiting the scope of candidate parameters for specialization. Similarly, Tick C [3] compiler generates optimized code at runtime. A significant improvement in performance is achieved, but the runtime code generation activity and optimizations incur a large overhead. In contrast, we minimize the runtime overhead by optimizing templates and generating specialized data at static compile time.

Some other dynamic code generation systems have been suggested in [5,15,16] that categorize the compilation process into different stages at which different optimizations can be performed. These models are effective enough to improve performance; however, they require the programmer intervention to decide which optimization could be appropriate at which stage.

In recent work related to dynamic optimization systems, the Dynamo [17] framework developed at HP Labs. can interpret the instruction stream. The hot traces of code are searched and a fragment cache is incorporated to apply dynamic optimizations. Another dynamic optimization framework, ADORE [7], is used to perform the dynamic cache prefetching. During execution, the hardware-based counters are used to compute the cache miss penalties that are reduced through prefetching. Similarly, the continuous compilation system [8] makes use of static and dynamic optimizations. The dynamic optimizations are continuously applied and are based on the analysis of code and performance. The continuous monitoring/profiling implemented in these systems incurs large overhead and thereby limits these approaches to the code having large number of invocations.

8. CONCLUSION

This article presents an optimization approach that is based on efficient instantiations of optimized templates. For complex benchmarks such as `ATLAS` and `FFTW`, we are able to achieve good speedup through this approach with a minimum increase in the code size.

The fast instantiations of the templates at static compile time and runtime reduce the overhead of runtime specialization. The optimized template is generated by specializing the code at static compile time. The static analysis and validation criteria are used to generate specialized data that reduce the overhead of iterative specialization. The runtime instantiations of the templates use specialized data to ensure the minimum possible overhead incurring during execution.

The cost of runtime code generation in this approach is far less than that in existing specializers and code generators. The optimizations on the template that are already performed at static compile time (due to specialization) bring significant improvement in performance.

DOI: 10.1002/cpe

REFERENCES

1. Muth R, Watterson SA, Debray SK. Code specialization based on value profiles. *Static Analysis Symposium*, London, U.K., 2000; 340–359.
2. Consel C, Hornof L, Marlet R, Muller G, Thibault S, Volanschi EN. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys* 1998; **30**(3es):19.
3. Poletto M, Hsieh WC, Engler DR, Kaashoek FM. 'c and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems* 1999; **21**:324–369.
4. Leone M, Dybvig RK. Dynamo: A staged compiler architecture for dynamic program optimization. *Technical Report*, Indiana University, 1997.
5. Grant B, Mock M, Philipose M, Chambers C, Eggers SJ. DyC: An expressive annotation-directed dynamic compiler for c. *Technical Report*, Department of Computer Science and Engineering, University of Washington, 1999.
6. Khan MA, Charles HP, Barthou D. Reducing code size explosion through low-overhead specialization. *Eleventh Annual Workshop on the Interaction Between Compilers and Computer Architecture*, Phoenix, U.S.A., 2007.
7. Lu J, Chen H, Yew PC, Hsu WC. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism* 2004; **6**:1–16.
8. Childers BR, Davidson JW, Soffa ML. Continuous compilation: A new approach to aggressive and adaptive code transformation. *NSF Workshop on Next Generation Software*, Nice, France, 2003.
9. Khan MA, Charles H-P, Barthou D. An effective automated approach to specialization of code. *Twentieth International Workshop on Languages and Compilers for Parallel Computing*, Urbana, IL, U.S.A., 11–13 October 2007.
10. Whaley RC, Dongarra J. Automatically tuned linear algebra software. *Technical Report UT-CS-97-366*, University of Tennessee, 1997. Available at: http://www.netlib.org/lapack/lawns/lawn131.ps [December 1997].
11. Calder B, Feller P, Eustace A. Value profiling. *International Symposium on Microarchitecture*, Los Alamitos, CA, U.S.A., 1997; 259–269.
12. Frigo M, Johnson SG. The design and implementation of FFTW3. *Proceedings of IEEE* 2005; **93**(2):216–231.
13. Makholm H. Specializing C—An introduction to the principles behind C-Mix. *Technical Report*, Computer Science Department, University of Copenhagen, 1999.
14. Grant B, Mock M, Philipose M, Chambers C, Eggers SJ. Annotation-directed run-time specialization in C. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (*PEPM'97*). ACM: New York, 1997; 163–178.
15. Consel C, Hornof L, Noël F, Noyé J, Volanschi N. A uniform approach for compile-time and run-time specialization. *Partial Evaluation. International Seminar*, Dagstuhl Castle, Germany. Springer: Berlin, Germany, 1996; 54–72.
16. Bala V, Duesterwald E, Banerjia S. Dynamo: A transparent dynamic optimization system. *ACM SIGPLAN Notices* 2000; **35**(5):1–12.
17. Khan MA, Charles H-P. Applying code specialization to FFT libraries for integral parameters. *Nineteenth International Workshop on Languages and Compilers for Parallel Computing*, New Orleans, U.S.A., 2–4 November 2006.

DOI: 10.1002/cpe