Applying Code Specialization to FFT Libraries for Integral Parameters

Minhaj Ahmad Khan and Henri-Pierre Charles

Université de Versailles Saint-Quentin en Yvelines

Abstract. Code specialization is an approach that can be used to improve the sequence of optimizations to be performed by the compiler. The performance of code after specialization may vary, depending upon the structure of the application. For FFT libraries, the specialization of code with different parameters may cause an increase in code size, thereby impacting overall behavior of applications executing in environment with small instruction caches.
In this article, we propose a new approach for specializing FFT code that can be effectively used to improve performance while limiting the code increase by incorporating dynamic specialization. Our approach makes use of a static compile time analysis and adapts a single version of code to multiple values through runtime specialization. This technique has been applied to different FFT libraries over Itanium IA-64 platform using icc compiler v 9.0. For highly efficient libraries, we are able to achieve speedup of more than 80% with small increase in code size.

1 Introduction

Modern optimizing compilers are able to trigger various optimizations if they are provided with the necessary information concerning variables used in the code. Most of the time, this information is not available until execution of the program. In this regard, code specialization can be used to expose a set of values to the program. But the impact of code specialization is diminished by the fact that specialization for a large number of values of a single variable can result in enormous code size increase.

For scientific and mathematical applications, different algorithms have been implemented in libraries that are able to calculate Discrete Fourier Transforms in O($nlogn$). These libraries provide support for DFT of different sizes with real and complex input data. Moreover, these code libraries are heavily dependent on integer parameters which can be fully exploited by the compiler if their values become known. However, it is difficult to specialize code with each possible value of the important parameter.

In this article, we propose an optimization technique that targets FFT libraries and makes use of code specialization in an efficient manner so that the problem related to code explosion is decreased. This technique makes an assumption that the optimizing compilers generate object code with minor differences for a large range of values of an integer variable. The basis of this assumption

is the fact that a compiler invokes a set of optimizations that is normally related to a range of values of a parameter. At static compile time, we perform a static compile time analysis to find out the instructions which are dependent on the specializing value in the object code. If these instructions fulfill the conditions required for our runtime specialization approach, a specializer is generated that modifies binary instructions for a new specializing value at runtime. This approach can therefore be acquired to limit number of versions required for specialization of code. The runtime specialization is performed with a small overhead (12 to 20 cycles per instruction) since we require specialization of only a limited set of binary instructions instead of generating complete code during execution.

The remaining part of the paper is organized as follows. Section 2 gives an example describing the behavior of specialization. The new approach of code specialization has been proposed in section 3, whereas section 4 elaborates it through description of an implementation framework. Section 5 presents the results obtained after specialization. The related work has been discussed in section 6 before concluding in section 7.

2 Motivating Example

The code specialization is used to reduce the number of operations to be performed by the program during its execution, however, it can also be used to facilitate compiler with necessary information required to optimize the code in an efficient manner. Different optimizations can be performed by the compiler such as loop unrolling, software pipelining, constant folding, dead-code elimination and better instruction scheduling.

For example, consider the following code:

```
void Function(int size, double * a, double * b, int dist, int stride){
     int i;
     for (i=0; i< size; i++, a+=dist, b+=dist){
          a[stride*2] = a[stride] + b[stride];
     }
}
```

Different value of specializing parameter may cause the compiler to generate different code. For a small value of loop size, loop can be fully unrolled and the loop overhead can be reduced. Similarly, if we specialize the loop size for larger value (e.g. 123), loop can be pipelined with partial unrolling depending upon the architecture for which the code is being compiled. The optimization sequence for all the specialized versions would therefore be different in all the cases depending upon the specializing value and this is the reason which impacts execution performance of an application.

Furthermore, if we specialize above given code with value of `stride`, then the compiler is able to determine the offset at static compile time and a better dependence analysis can now be performed to invoke more optimizations. For

the above given example, when compiled with `icc v 9.0`, the loop is partially unrolled and software pipelining is performed with a pipeline depth of 2. For original unspecialized version, no software pipelining is performed. This makes the code execution faster for specialized version. However, keeping different versions specialized with all possible stride values would degrade performance of application. Therefore an attempt is made to keep single version and perform dynamic specialization to adapt this version to a large range of values. When we specialize the code with `stride` value being 5 and `stride` value 7, the object code produced after compilation would be similar in terms of optimizations and would differ in immediate constants that would be based on `affine` formula of the form :

$$Immediate_value = stride * A + B. \tag{1}$$

This formula can then be used during execution to adapt single version to a large number of values, thereby taking full advantage of optimizations at static compile time instead of performing heavyweight optimizations at runtime. The task of specialization of binary instructions can be accomplished through efficient dynamic specializer as proposed in next section.

3 Approach for Limited Code Specialization

Existing compilers and partial evaluators [17, 1] are able to perform partial evaluation if different values in the program become known. A small set of values and parameters therefore needs to be selected which should be important enough to have some impact on execution speed. Our current approach of code specialization targets only integer parameters and it fully conforms with FFT libraries where the code optimizations depend largely on integer parameters. So far our approach is restricted to positive integral values to keep the semantics of the code after specialization.

A profiling analysis can be performed to collect information regarding most frequently used functions and their parameter values. The hot parameters can then be specialized statically through insertion of wrapper (for redirection) describing the version with the parameter replaced by its value in the function body. However, for an integer parameter with n-bit size, we will require 2^n versions resulting in huge increase of code size. Therefore, we find a generic template specialized at static compile time and adapt it to different versions during execution through invocation of a runtime specializer. The generic template is highly optimized at static compile time since the unknown value has been made available to the compiler by providing a dummy value for the specializing parameter. This approach is different from those proposed in [1–4] which perform partial evaluation for code that has either already been specialized or suspend specialization until execution.

The main steps for our code specialization are described as follows:

1. Insert a wrapper at the start of function body containing a call to specializer and to maintain software cache. This would be used to reduce the number

of versions to be generated during execution. To define range R, initialize MINVAL to be 1 and MAXVAL with the maximum value for the type of specializing parameter.

2. Specialize the code with different positive values of the integer parameter and after compilation obtain versions (at least two) of assembly code (or dumped object code) generated with full optimizations. These versions contain the templates for which the runtime specialization will be implemented.
3. Analyze the assembly code to find the differences b/w two versions generated in previous step. The instructions will differ in immediate values which are based on the affine formula of the form $a * param + b$. Annotate all such instructions which differ in two versions. If instructions differ in any literal other than the immediate operand or they do not conform to the above given formula, go to last step.
4. Against each annotated instruction in the generated versions, solve the system of linear equations to calculate the values of a and b with constraints $param, a > 0$.
5. The analysis of formulae would reveal information regarding the range of values for which the template would be valid and modify the source code (MINVAL and MAXVAL) to contain this range. The range (R) can be calculated as follows:
   Let $a_i * param + b_i$ be the formula generated for i-th instruction, then we have,
   $$S_i = MAX\left(1, \left\lceil \frac{1 - b_i}{a_i} \right\rceil\right), \text{ and } E_i = \left\lfloor \frac{ARCHMAX_i - b_i}{a_i} \right\rfloor,$$
   where $ARCHMAX_i$ represents maximum value that can be used as immediate operand for i-th annotated instruction. The new range R for *param* with S = $\{S_i$, for i=1 to n$\}$ and E=$\{E_i$, for i=1 to n$\}$, can be represented as:
   $$param \in [MINVAL, MAXVAL], \text{ where,}$$
   $$MINVAL = MAX(S) \text{ , and } MAXVAL = MIN(E).$$
6. Find out the exact locations where the immediate values differ in two versions, and as a consequence, the runtime specializer can be generated that will modify the binary instructions in the template. The runtime specializer requires information regarding starting location, offsets of the locations to be modified and the formulae to calculate new values. The assembly code version and the specializer generated need to be linked to make invocation of specializer possible during execution of program.
7. For the values for which the assembly code versions do not fulfill the conditions, or it is required to keep static version (through programmer directives), define a static value in the source code for the specializing parameter by generating another version of the original function code. Modify the wrapper and the range in the function code to contain the branch to redirect to static specialized code.

Let T be set of values for which the static specialization needs to be performed, then, we need to modify the range R as:

$$R = R - T.$$

During execution, the wrapper inserted inside the code would cause the runtime specializer to be invoked prior to the execution of the specialized code for a particular value. When the same function code is called next time with the same value of specializing parameter, the wrapper would redirect the call to specialized code thereby avoiding the call to specializer and minimizing runtime overhead.

4 Implementation of Code Specialization

The low-level runtime specialization is a complex task and it would be cumbersome for programmers to manually perform specialization at binary code level. In addition, they might need to perform comparison of instructions, generation of formulae for all instructions which differ and find the valid range by taking into consideration the formats of binary instructions for that architecture. To automate this specialization technique, a small software prototype has been developed that incorporates source code modification, specializing invariants analysis and runtime specializer generation as depicted in Figure 1.

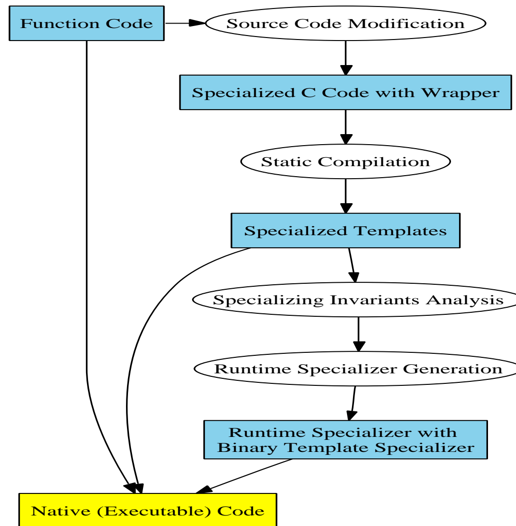

Fig. 1. Automation of Runtime Specialization

4.1 Insertion of Wrapper and Specialization

The source modification involves searching for the integer parameters in the function and replacing them with a constant value. Different functions can be specialized with different parameters. For our experiment, it is necessary to obtain the assembly code versions after specialization which resemble in terms of optimizations and differ only in a fixed set (*add*/*mov*/*cmp*) of instructions. Therefore, the prime numbers are preferred to be used as specializing value to obtain such versions of assembly code which avoid any value specific optimizations. The source code is modified to contain other versions of the candidate function specialized with an integer value. Moreover, in the function code, a wrapper is inserted containing the branch to specializer invocation only in the case when the function receives a new value. If the function receives the same value of the specializing parameter, the redirection code would cause the specialized function to be directly invoked avoiding call to specializer. The pseudo-code for the wrapper is shown in Figure 2.

```
Function (... int P2, int P3, int P4, ...)
  if P3 value in [MINVAL, MAXVAL]
    if the code was specialized with a different P3 value
      Specialize function with P3
    end if
    call Specialized function (..., P2, P4, ...)
  else
    call Standard code (... P2, P3, P4, ...)
  end if
```

Fig. 2. Source code after modification

The temporary values used for specialization need to be selected carefully. Usually smaller values are more suitable for specialization, and with `icc` compiler, the impact of specialization in terms of optimizations decreases with an increase in these values. The specialized assembly code is also the one to be linked to the client application in the final phase of static compilation. This specialized code contains templates with dummy values which would be replaced by new values and this is the task performed by automated runtime specializer.

4.2 Runtime Specializer Generation

The automation prototype includes a Specializing Invariants Analyzer (SIA) which performs analysis at static compile time and is used to prepare information required by runtime specializer. The analyzer finds the differences and then passes these differences to code specializer generator. Moreover, the locations of differences found will act as template holes to be filled, whereas, formulae on which these values are based, will be used to generate new values to adapt to.

The Specializer Generator takes this information and generates a specializer that will be linked to the specialized code to make modification during execution of the application.

After modification of source code, the code is compiled to either assembly code or dumped object code[1]. At least two versions are generated with different values of the specializing parameter. If we compare these versions, they would differ in some instructions containing constant values as operands. For example, for the above given example, the code generated by `icc v 9.0` compiler for code when specialized with the value $stride = 5$ and the one generated for $stride = 7$ would differ as shown in Figure 3. The comparison of the assembly code is

```
//With Stride = 5
  { .mii
      sub r24=r30,r19
      add r11=40,r28
      nop.i 0 ;;
  }
  { .mib
      add r14=20,r24
      nop.i 0
     (p9) br.cond.dptk .b1_3;;
  }
    ....
```

```
//With Stride = 7
  { .mii
      sub r24=r30,r19
      add r11=56,r28
      nop.i 0 ;;
  }
  { .mib
      add r14=28,r24
      nop.i 0
     (p9) br.cond.dptk .b1_3;;
  }
    ....
```

Fig. 3. Assembly Code generated by icc v 9.0

performed by the SIA, and formulae are generated on the assumption that in both versions, these formulae are of the form:

$$Val_1 = Param_1 * a + b, \text{ and } Val_2 = Param_2 * a + b.$$

where Val_1 and Val_2 are immediate operands of binary instructions, and, $Param_1$ and $Param_2$ are the values of the parameter with which the versions are specialized. Solving these two equations provides us the values of a and b. The formulae with known a and b are generated at static compile time and during execution only new value of actual specializing parameter ($Param$) is passed as shown in Figure 4. After solving equations, it is possible to find the range for which the code will be valid. For example, an `adds` instruction over Itanium allows a 14-bit signed immediate value to be used as operand, thereby making the above given code valid for range (i.e. $MAXVAL$) up to 1023.

For specialized code with $stride = 8$, runtime specialization will modify the immediate constants in Figure 3 with new values 64 and 32. This approach of runtime specialization is highly efficient and can be used to adapt a single version to a large range of stride values.

[1] Using `objdump` utility makes the code offset locations easy to calculate and it also facilitates to resolve different pseudo-code to actual binary instructions

Binary Template Specializer With the information of formulae and locations for each instruction after analysis at static compile time, a specializer is generated to invoke *Binary Template Specializer* as given in Figure 4. For Ita-

```
void Specializer_Function(long base_address, register long param)
        BinaryTemplateSpecializer( base_address, 6, 1, param * 8 + 0)
        BinaryTemplateSpecializer( base_address, 7, 0, param * 4 + 0)
             ..........
```

Fig. 4. Invocation of Binary Template Specializer

nium where a single bundle contains 3 instructions, we also require the location of instruction within the bundle since 128-bit binary bundle contains constants at different locations of the instruction word depending upon its location in the bundle. Therefore, a different specializing strategy is adopted for each instruction location in the bundle.

Runtime Activities The runtime specializer may include some initialization steps (e.g. on Itanium to make the segment of code modifiable). After the initialization, the specializer is invoked which in turn makes multiple calls to the Binary Template Specializer passing it the information regarding the location of instruction and new immediate value. This information causes the Binary Template Specializer to fill the binary templates with new values. The specializer may be invoked multiple times depending upon the number of instructions, however, it incurs a small overhead due to specialization of a limited set of instructions instead of complete function code. Finally, we also need to perform activities for cache coherence such as flushing and synchronization.

4.3 Static Versioning

In cases where the object code does not conform to conditions specified for dynamic specialization, we can statically specialize the code. We require the use of following directive to proceed for static specialization.

#defspec *param* `val`

This directive would cause the code modifier to generate a version together with the insertion of wrapper to redirect execution to this version when the *param* receives the specializing value `val`. The runtime overhead in this case would be reduced to single branch, but at a cost of increase in code size.

5 Implementation Setup and Results

Different FFT libraries including FFTW, GSL FFT, Scimark, FFT2 have been optimized through our algorithm on Itanium IA-64 processor making use of Intel

compiler `icc v 9.0`. These libraries have been compiled with `-O3` parameter for optimization. The performance measurements are made using `pfmon` library for computation of 1-dim DFT for complex input data.

It is to be noted that the runtime specialization is very efficient and requires 12 to 20 cycles for each binary instruction to be specialized. Most of the times, we do not require multiple invocations to amortize overhead and this makes our runtime specialization algorithm more efficient than those implemented in [1, 5, 6, 3] where break-even point is very high (up to 100). In case where the code is already specialized for hot parameter, the runtime specialization for less effective parameter can deteriorate performance of the application.

The average specialization overhead with respect to execution time of the application and code size increase together with average speedup obtained of all executions, have been shown in table 1.

	FFTW	GSL-FFT	Scimark	FFT2	Numutils	Kiss-FFT
Avg. Specialization Overhead (w.r.t exec.time)	1.00%	1.00%	2.00%	2.00%	1.00%	2.00%
Code Increase (w.r.t total library size)	10.00%	2.00%	19.00%	10.00%	12.00%	11.00%
Avg. Speedup	23.26%	11.85%	17.53%	7.74%	3.38%	6.29%

Table 1. Performance Summary

5.1 FFTW

FFTW [7, 8] is one of the fastest libraries able to compute DFTs of real and complex data in $O(n \log n)$ with any size of n. The library contains large number of codelets which were specialized both statically and dynamically. The codelets required 4 to 20 binary instructions to be modified during execution. The FFTW wisdom was generated in `exhaustive` mode for calculating out-of-place 1-dim complex DFTs having size of powers of 2. The graph in Figure 5 shows that

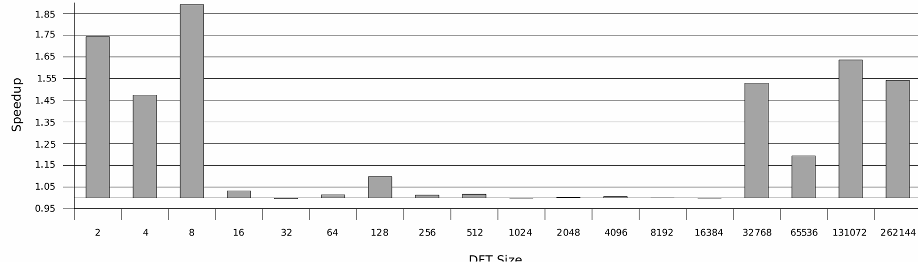

Fig. 5. FFTW Library Specialization Results

the large speedup occurs for small values of N (input array size) for which the codelets are invoked with small loop counter and stride values. With the known strides in smaller codelets e.g. $n1_2$, the Initiation Interval (the number of cycles required for an iteration in stable mode of pipeline) for the specialized version was reduced. Similarly, with the small loop trip count, the compiler fully unrolled the loop to produce large speedup for 8. For large values, the codelets are invoked multiple times resulting in an accumulated speedup through invocation of dynamically specialized code.

5.2 GSL FFT Library

The GNU Scientific Library (GSL)[9] is a numerical library containing programs able to solve different mathematical problems including FFT, BLAS, Interpolation and Numerical Differentiations etc. The figures 6 and 7 show speedup obtained with calculation of complex (forward) DFT of size of different prime numbers and those of powers of 2. The functions `fft_complex_pass_n` and `fft_complex_radix2_transform` were specialized with stride value to be one. For the function `fft_complex_pass_n`, the compiler had generated more data cache prefetch instructions and pipelined loops than those in the standard version. For the function `fft_complex_radix2_transform`, the object code generated by *icc* was almost similar for both the specialized and unspecialized versions. This shows that our specialization technique is heavily dependent on the optimizations done by the compiler at static compile time.

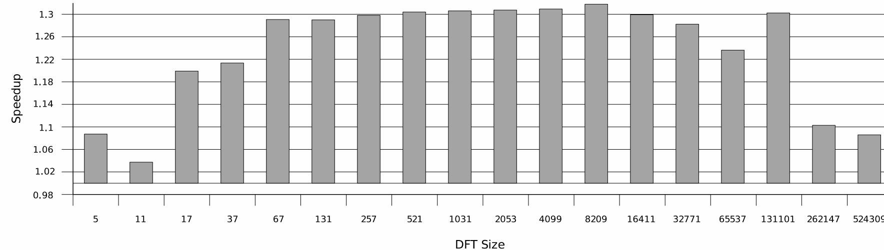

Fig. 6. GSL-FFT Specialization Results(Prime Numbers)

5.3 Scimark2 Library

SciMark 2.0 [10] has been developed at the National Institute of Standards and Technology (NIST). Its C source function `fft_transform_internal` was specialized to generate a binary template that required 7 binary instructions to be specialized during execution. The small value of specializing parameter let the compiler remove the overhead of filling and draining the pipelines, however

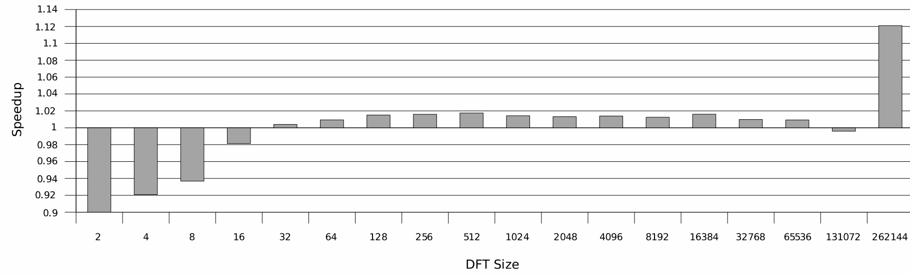

Fig. 7. GSL-FFT Specialization Results (Powers of 2)

the code was restricted to a limited set of values (up to 128). In contrast to small values, the code produced by compiler after specializing with large static values did not impact performance to a large factor and resembled with the original object code.

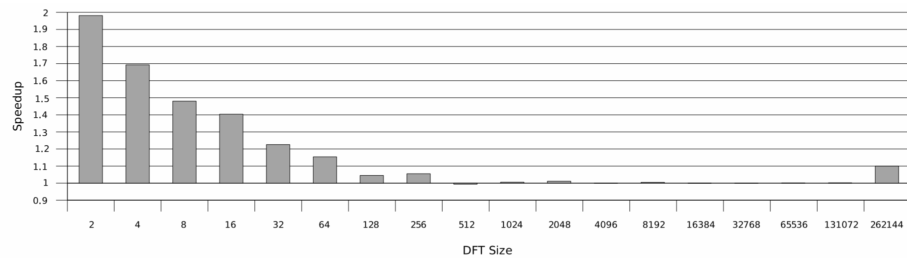

Fig. 8. Scimark Library Specialization Results

5.4 FFT2 Library

The FFT2 library[11] contains routines to perform Cooley-Tukey Fast Fourier Transform on complex and real samples. The dynamic specialization for the `join` function was performed for parameter m. The binary template was generated which contained 5 binary instructions to be modified. The static specialization was limited to 2 versions with m=1 and m=2 which are frequently called with other different values for which dynamic specializer was invoked. The Figure 9 shows speedup only for large values where the function is repeatedly called with the same value as in case of large sizes of input array. The code for static versions contained pipeline with large depth making it suitable for large executions. Moreover, for small function calls, the performance degrades due to overhead of dynamic specialization caused by recursive calls of the same function with different input values.

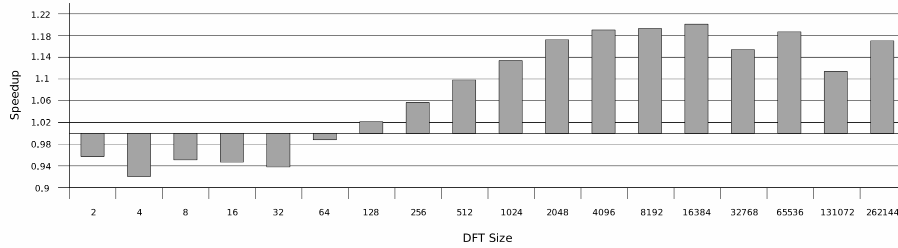

Fig. 9. FFT2 Library Specialization Results

5.5 NumUtils Library

The NumUtils library[12] contains functions to solve different mathematical problems. The results in Figure 10 show the speedup obtained through different versions of routine `fft` since a generic template could not be obtained after invariants analysis. The static specialized code however contained less number of loads and stores sufficient enough to produce speedup.

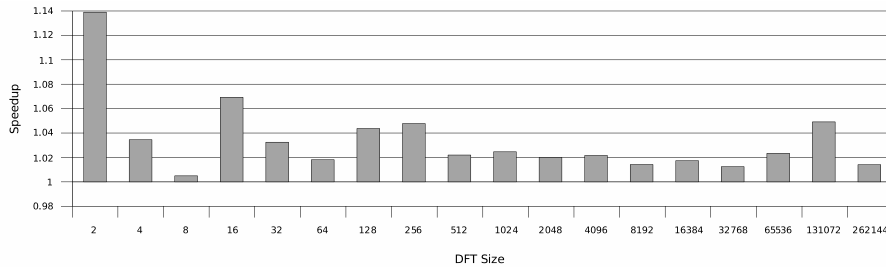

Fig. 10. NumUtils Library Specialization Results

5.6 Kiss-FFT Library

For the library Kiss-FFT[13], the functions `kf_bfly2` and kf_bfly4 were specialized with versions for m=1 and m=1,2,4 respectively. Moreover, the function kf_bfly4 was dynamically specialized as well generating a template having 7 locations to be modified during execution. The binary template to be specialized and statically specialized versions contained very less number of memory operations as compared to unspecialized code. Moreover, the repeated invocation of these functions resulted in speedup as shown in Figure 11.

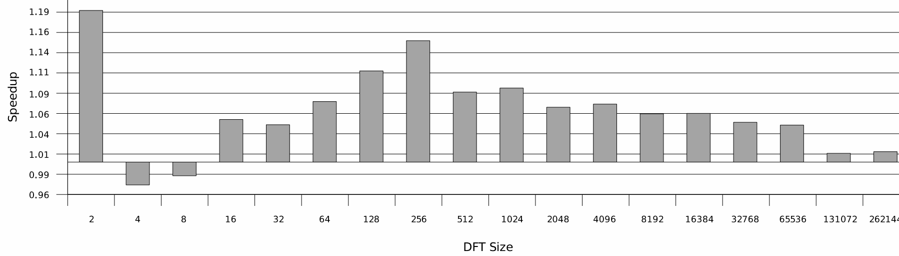

Fig. 11. Kiss-FFT Library Specialization Results

6 Related Work

The code specialization and dynamic compilation have been applied to a diverse domain of applications including the image filtering, geometrical transformations, JPEG compression, operating systems, pattern matching and DBMS Pattern Recognition [14, 15, 1, 3, 16, 17, 4, 18, 5]. Some systems (e.g. C-Mix [17]) perform source to source transformation to generate code after partial evaluation at static compile time, while others are able to perform specialization during execution of the program through runtime code generation. Most of these approaches perform specialization and optimizations in two stages. In first stage, a high level static compile time analysis is performed and optimizations are performed only for known values, thereby leaving other optimizations to be performed during execution of the code. On the contrary, our approach relies on low-level analysis of code to fulfill the required criteria. Also, these code generators perform large number of optimizations during execution at the cost of overhead, whereas our technique is able to obtain highly optimized code at static compile time.

The DCG [19] is a retargetable dynamic code generator and makes use of intermediate representation of lcc compiler to translate into machine instructions through BURS (Bottom-Up Rewrite System). For different mathematical functions, it is able to achieve large speedup (3 to 10) but at a cost of 350 instructions per generated instruction, making it suitable only for applications requiring minimum dynamic code to be generated during execution. Similarly, Tick C [6] compiler can generate code during execution and perform optimizations including loop unrolling, constant folding and dead-code elimination. With Tick C, a speedup up to 10 has been achieved for different benchmarks, but it requires multiple calls to amortize overhead. With VCODE [20] interface, optimized code is generated during execution of the program at the cost of 300 to 800 cycles per generated instruction. The optimized code is generated for single value, thereby requiring the code generation activity to be invoked again for other specializing value. In contrast, since we are constrained to apply single transformation during execution, we are able to reuse the same binary code for multiple specializations.

Fabius [21, 3, 22] is the system that is able to generate specialized native code at runtime for programs written in Standard ML. Although the average overhead

(6 cycles per generated instruction) is less than that incurred through our approach, they require complete function code to be generated during execution. This is different from our specialization approach where we require only a limited number of instructions to be specialized.

Tempo [2, 1] is a specializer that is able to perform offline and online partial evaluation of code. It performs a large static compile-time analysis to specialize the code during execution by keeping different versions of the same code specialized with different values. For runtime generation of code and optimizations it invokes Tick C compiler. With dynamic specialization of FFT, they are able to achieve average speedup[2] of 3.3 , with break-even point up to 8 and requiring separate version for each size of DFT. The approach adopted by Tempo differs from ours in that if the code is not already specialized, all the optimizations will be performed during execution, whereas, our approach benefits from optimizations at static compile time by exposing constant values of specializing parameter.

7 Conclusion and Future Work

The exposure of values of different parameters enables the compiler to generate more efficient code. It is possible to limit the size of specialized versions through implementation of an efficient dynamic specializer which is able to specialize a fixed set of binary instructions during execution. In this way, the overhead of code generation and memory allocation is minimized, and a good speedup can therefore be obtained. For dynamic specialization, we first generate templates after a low-level analysis of code. The templates should contain instructions with immediate operands which must be dependent on the value of specializing parameter. Once a version is specialized with dynamic value, we do not require to invoke specializer for the same input value. However, if the specializer is called multiple times with different values, or the code generated at static compile time is not highly optimized, then the specialization may deteriorate the performance.

A cost analysis is being incorporated in conjunction with dependence test to automate the decision of when to apply dynamic specialization. Currently this approach makes use of specializers which are specific to Itanium architecture. So, we intend to generalize the specializer generation and template code specialization for multiple platforms.

References

1. 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)
2. Nol, F., Hornof, L., Consel, C., Lawall, J.L.: Automatic, template-based run-time specialization: Implementation and experimental study. In: International Conference on Computer Languages (ICCL'98). (1998)

[2] Results show only 3 invocations with input size 32, 64 and 128

3. Leone, M., Lee, P.: Dynamic Specialization in the Fabius System. ACM Computing Surveys **30**(3es) (1998)
4. Consel, C., Hornof, L., François Noël, Noyé, J., Volanschi, N.: A uniform approach for compile-time and run-time specialization. In: Partial Evaluation. International Seminar., Dagstuhl Castle, Germany, Springer-Verlag, Berlin, Germany (1996) 54–72
5. 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)
6. 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** (1999) 324–369
7. Frigo, M., Johnson, S.G.: The design and implementation of fftw3. In: In proceedings of IEEE, Vol. 93, no. 2. (2005) 216–231
8. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing. Volume 3., Seattle, WA (1998) 1381–1384
9. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Booth, M., Rossi, F.: GNU Scientific Library Reference Manual - Revised Second Edition. http://www.gnu.org/software/gsl/ (2005)
10. Gough, B.: SciMark 2.0. http://math.nist.gov/scimark2/ (2000)
11. Valkenburg, P.: FFT2 Library. http://www.jjj.de/fft/ (2006)
12. Hein, C.: Numerical Utilities. atl.external.lmco.com/projects/csim/xgraph/numutil (2003)
13. Borgerding, M.: KissFFT v1.2.5. http://sourceforge.net/projects/kissfft/ (2006)
14. Jones, N.D., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall International, International Series in Computer Science (1993)
15. Brifault, K., Charles, H.P.: Efficient data driven run-time code generation. In: LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems, New York, NY, USA, ACM Press (2004) 1–7
16. Locanthi, B.: Fast bitblt() with asm() and cpp. In European UNIX Systems User Group Conference Proceedings (1987)
17. Makholm, H.: Specializing c- an introduction to the principles behind c-mix. Technical report, Computer Science Department, University of Copenhagen (1999)
18. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: Annotation-Directed Run-Time Specialization in C. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), ACM (1997) 163–178
19. Engler, D.R., Proebsting, T.A.: DCG: An efficient, retargetable dynamic code generation system. In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California (1994) 263–272
20. Engler, D.R.: Vcode : A retargetable extensible, very fast dynamic code generation system. In: In Proceedings of the SIGPLAN 96 Conference on Programming Language Design and Implementation, ACM, New York. (1996)
21. Leone, M., Lee, P.: Optimizing ml with run-time code generation. Technical report, School of Computer Science, Carnegie Mellon University (1995)
22. Leone, M., Lee, P.: A Declarative Approach to Run-Time Code Generation. In: Workshop on Compiler Support for System Software (WCSSS). (1996)