Reducing code size explosion through low-overhead specialization

Minhaj Ahmad Khan (mik@prism.uvsq.fr) Henri-Pierre Charles (hpc@prism.uvsq.fr)
Denis Barthou (bad@prism.uvsq.fr)
University of Versailles, France

Abstract

The overhead of performing optimizations during execution is the main hindrance in achieving good performance for many computation intensive applications. The runtime optimizations are required due to lack of precise information at static compile time. In this article, we describe a new optimization method, the hybrid specialization, able to overcome this problem to a large extent.

Specialization proceeds by exposing values for a set of candidate parameters at static compile time. The next part of specialization is performed at runtime by a dynamic specializer which is capable of reusing optimized code and specializing binary code generated in the last step. The dynamic specializer is generated automatically after validating code against the required criteria. It can be used to adapt binary code to different values and therefore reduces the possible runtime code generation. Moreover, it incurs a small overhead for runtime specialization as compared to that incurred in existing dynamic code generators or specializers.

Initial results over Itanium-II architecture show improvement in performance for different benchmarks including `SPEC CPU2000`*,* `FFTW3` *and* `ATLAS`*.*

1 Introduction

Code generated statically by state-of-the-art compilers very often does not reach the performance level of hand-tuned code. One of the reasons is that a static compiler misses a large part of the application context, and must be conservative concerning the possible values that variables can take during execution.

Profile-guided optimization is a known approach to generate higher performance code, resorting to the analysis of previous runs on training input data sets. This kind of iterative/continuous compilation process[2, 4] (compilation, run, run analysis) is able to catch the application behavior, by specializing code fragments to particular values. This specialization, also called *versioning* comes at the expense of code expansion. This drawback limits versioning to be performed for a few values. Run-time specialization, on the other hand, does not suffer this limitation.

```
void Function(float *A, float *B, int size, int stride){
    int i;
    for (i  = 0; i < size; i++)
        A[i*stride] = B[i] + A[i];
}
```

Figure 1. Motivating example

The run-time specialization systems[11, 15, 12, 16, 23, 21] and off-line partial evaluators[5, 7] are used to generate code during execution of the program. Most of them keep different versions of the code both during static compile time and during execution. These specializers generate code and perform optimizations whenever a parameter receives a new input value. Invocation of a compiler for optimizations (partial evaluation, scheduling, software pipelining), results in a large amount of overhead which needs to be compensated by multiple calls of specialized code. For the specialization approach proposed in this paper, we do not require such heavyweight activities and limit the run-time code generation to modification of a small number of binary instructions. Moreover, information regarding these instructions can be fully computed at static compile time.

Consider the code in Figure 1 which mimics a codelet from `FFTW` library. Specialization of this function consists of the replacement of one or more formal parameters by a constant value. From a compiler perspective, turning a parameter into a constant may enable several aggressive optimizations. Lack of information concerning parameter *stride* constraints the compiler to assume that it may equal 0, generating dependence from one iteration to the next. Specializing *stride* with value 5 for instance implies that at least 4 successive iterations are independent. Software pipelining optimization can efficiently take advantage of this information to increase the instruction level parallelism.

Since such values are available only during execution for most real-life applications, one solution would be to perform at run-time the dependence analysis and optimiza-

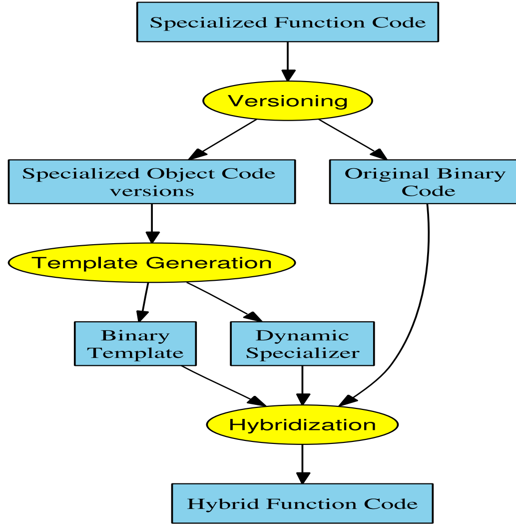

Figure 2. Overview of Hybrid Specialization

tion (software pipelining in this case), together with heavy-weight code generation. This would certainly require hundreds of invocations of the code to amortize this overhead. This situation gets even worse if the parameter values change frequently and entire optimization and code generation activities are iterated.

Our method is based on the observation that the same code, specialized for a stride value of 7 for instance, may result in nearly the same binary code, with exactly the same schedule (due to resource constraints). Identifying the differences between the two versions specialized for 5 and 7 (coming from the dependence distance) can therefore lead to the generation of a more general version, a template, that can be instantiated into a pipelined code for values 5, 7 and possibly many others. The differences between the binary versions depends on the code and on the compiler.

Based on the assumption that compiler generated versions are similar for a large range of values, we propose an intermediate solution between pure static and pure dynamic specialization of code, termed as *Hybrid Specialization*. This approach incorporates dynamic specialization to be applied on code (template) which is specialized and generated at static compile time, and versioning to be performed for the cases where the template could not be generated. So runtime activities require the specialization of a few number of binary instruction parameters. Since the template is generated at static compile time, it is highly optimized and assumed to work better than the original unspecialized code. Consequently, we get the best of the two worlds: efficient and fast code specialization at runtime and aggressive optimizations at static compile time.

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

2 Template Creation and Legality Requirements

This section formalizes the notion of template used for dynamic specialization, and then, infers the legality conditions.

Consider the code of a function F to be optimized, we assume without loss of generality that F takes only one integer parameter X. By versioning F with two values v_1 and v_2, we obtain two functions at object code level, F_{v_1} and F_{v_2} resp., that are assumed to perform better than the original code. Moreover, these versions must contain constants (at instruction level) as given follows:

$$C_i^{v_1} \in (F_{v_1}) = \alpha_i.v_1 + \beta_i,\ \forall\ i \in \{1..p\} \quad (1)$$

$$C_i^{v_2} \in (F_{v_2}) = \alpha_i.v_2 + \beta_i,\ \forall\ i \in \{1..p\}. \quad (2)$$

Only immediate values C_i of p instructions differ from one instantiation to the other with all α_i and β_i being constants. Now, given two binary versions, F_{v_1} and F_{v_2} with such instructions, we generalize them into a binary template to be instantiated with V at run-time. For a small set of values R for which code is valid, we require to modify p binary instructions to contain constants of the form:

$$C_i^V = \alpha_i.V + \beta_i,\ \forall\ i \in \{1..p\},\ V \in R. \quad (3)$$

For n parameters, we can generalize the criteria with p instructions to contain run-time values of the form:

$$C_i^{V_k} = \sum_{k=1}^{n} \left(\alpha_i.V_k + \beta_i\right),\ \forall\ i \in \{1..p\},\ V_k \in R_k. \quad (4)$$

If the versions satisfying Equations 1 and 2 are found, the range R_k is computed as follows:

- If X is loop bound and both bounds are constant, loop unrolling may generate different codes according to X. For example, partial loop unrolling of the loop by a factor of 4, generates no tail code when X is a multiple of 4, whereas, there may be a two iteration tail loop for $X = 2 \bmod 4$. The legal range in this case is expressed as a condition, modulo the unrolling factor.

- If X is involved in a condition, the compiler may perform some dead-code elimination when the value of

X is known. The legal range is determined by the values for which the condition is true (or not). Failure to compute this range statically would mean that dynamic specialization is unsafe.

- Limit the range of parameter values in order to keep all assembly instructions legal. Indeed, some assembly instructions are only valid for a defined set of immediate values (e.g. post-increment of address registers defined with 6 bits). Since all formulae generating these instructions are affine, we must find the range by calculating the maximum and minimum possible values for each instruction (in Equation 4) followed by calculating maximum and minimum for entire code of the function.

3 Approach of Hybrid Specialization

Similar to other specialization strategies, hybrid specialization is also guided through profiling information and proves to be beneficial when applied to hot regions of code. Once the information regarding a set of intervals of values of a parameter [3, 14] becomes available, the hybrid specialization proceeds as given below:

1. Versioning of selected functions;

   Different specialized versions of the selected function are generated after parsing the code and replacing the parameter with constant values. These versions will be used in both static specialization and dynamic specialization.

2. Analysis of assembly code within intervals for template creation;

   To proceed for dynamic specialization, we need to generate one specializer and single template for each interval. Therefore multiple versions (of assembly code /dumped object code) are searched within one interval to conform to the conditions given in section 2. These versions must differ only by some constants. We assume that the formula used by the compiler to generate these constants from the parameter value is an affine function. This ensures that the necessary overhead to evaluate the template at run-time is kept small. The formulae are generated assuming them to be of the form: $v = \alpha \times param + \beta$, where α and β are constants and $param$ is the value of specializing parameter.

   For n parameters with p binary instructions to be specialized, the formulae of the form $v_k = \alpha_{ki} \times param_i + \beta_k$ for $1 \leq i \leq n$ and $1 \leq k \leq p$, are solved in $O(n^3 p)$ at static compile-time.

3. Using the versions found in previous step, generation of runtime specializer (if possible);

   If the system of equations can not be solved, i.e. required *consistent* versions are not found for an interval, then dynamic specialization does not proceed further, since no generic template could be found. In the other cases, when the equations are solved and constants α and β are found for each of p instructions in the object code versions, then specializer generation takes place with a valid range. This range is calculated by validating code against the criteria defined in Section 2. Subsequently, starting location, offset of binary instruction to modify and formula to calculate new value are gathered. Based upon this information, the self-modifying code must then be generated together with code for cache coherence. We make use of a lightweight runtime *Instruction Specializer* that accomplishes the task of binary code modification (only p binary instructions) in an efficient manner.

4. Hybridization, i.e., putting statically specialized versions, the template code with the dynamic specializer (if generated through last step) and the initial code altogether for the hybrid version;

   To limit number of specialized versions for a function, we take into account the number of valid templates found. So this number of templates will actually reduce the number of static versions.

4 Implementation Framework and Experimentation

The hybrid specialization approach (depicted in Figure 2) has been implemented in HySpec framework to specialize function parameters of integral data types. These parameters should be defined through a directive of the form :

#pragma specialize *paramName*[=*interval*,...]

When the interval is not given, specialization would occur after obtaining value profile through instrumentation[1] for integral parameters. Otherwise, the interval can be explicitly defined based on application-knowledge.

For hybrid specialization of code, HySpec can perform various activities including: parsing and instrumenting code for specialization, analysis of object code(dumped/assembly) versions, finding differences among these versions followed by generation of runtime specializer (by solving equations) containing formulae, and the hybridization of the versions according to the specified criteria. In this section, we describe details of each of these steps over Itanium-II architecture.

[1]HySpec supports instrumentation of code at routine level for value profiling integral parameters and forming intervals

4.1 Versioning and Invariant Analysis

A specialized version of a function is generated by defining the value (taken from interval) for the parameter. Moreover, the code of the function is instrumented to redirect control to the specialized versions. For each interval selected, dynamic specialization proceeds subject to the generation[2] of *consistent* binary code versions. These versions are compared instruction-wise and should differ only in a limited set of binary instructions with immediate constants.

For the example (Figure 1) considered, bundles of the code generated by `icc` compiler, when specialized with the value *stride*=5 and the one generated for *stride*=7 differ only in some constants in the object code as shown in Figures 3(a) and 3(b) respectively. These instructions correspond

```
add r9= 5, r57
stfs [r2]= f43, 20
add r37= 495, r56
......
```
(a) `stride=5`

```
add r9= 7, r57
stfs [r2]= f43, 28
add r37= 693, r56
......
```
(b) `stride=7`

Figure 3. Assembly code generated by `icc` v9.0

to address computation relative to the stride. A comparison of the entire assembly code versions is therefore performed, and formulae are generated after solving the system of equations, assuming them to be of the form: $v = \alpha \times stride + \beta$. The invariant analysis comprises search for such equivalent code versions conforming to the conditions of template generation given in Section 2, and consequently, the legality range for the template is validated.

4.2 Runtime Specializer Generation

After checking code correctness/equivalence for a range of values, we are able to define a template. To instanciate the template during execution, a type of self-modifying code called binary template specializer (Figure 4) is generated. In order to modify binary instructions, the template specializer is provided with the basic information regarding starting location of the code, and the parameter to specialize with. It contains multiple invocations of *Instruction Specializer* with necessary information such as the offset of the instruction (bundle number and instruction number within bundle for Itanium-II) to modify and the formula which will produce the new value during execution.

During execution, all the binary instructions at corresponding template locations are filled with new values generated through affine functions. A runtime view of the template code when specialized with $stride$=8 has been shown

[2]Compilation environment variables should bet set

```
void BTS(Function_Address, int param){
    ISpec(Function_Address, 14, 0, param * 1 + 0)
    ISpec(Function_Address, 16, 0, param * 99 + 0)
    .........
}
```

Figure 4. Binary Template Specializer

```
add r9= 8, r57
stfs [r2]= f43, 32
add r37= 792, r56
......
```

Figure 5. Post-Specialization view

in Figure 5. The generated specializer also performs activities for cache coherence as required by Itanium architecture for the instructions at memory locations modified during execution.

4.3 Hybridization With Bounded Static Versioning

To bound the number of specialized static versions, we make use of a heuristic that bounds statically specialized versions depending upon the number of valid templates found. For a parameter with b-bits, let D be the number of intervals for which valid templates are found (for dynamic specialization), then we bound static specialization to $b/4 - D$ versions. This heuristic works well not only for the parameters with large variance in their values, but also for the parameters which have a small number of different values. Figure 6 shows the wrapper generated for the final code. It contains branches to redirect execution control to statically specialized code, invocation of specializer and dynamically specialized code.

```
void Function(float * array, int size,int stride){
  if( stride == 1){
    Function_stride_1(array, size);
    return;
  } else if (stride >=MINVAL and stride <= MAXVAL){
    specializer(stride);
    Specialized_Function(array, size);
    return;
  }
  else
    /* Original code ...  */
}
```

Figure 6. Pseudo-code for hybridized wrapper

5 Experimental Results

This section presents the detailed results of hybrid specialization of integral parameters [3] applied to a large number of functions from ATLAS, FFTW3 and SPEC CPU2000 benchmarks. To profile for frequently used parameter values, code is instrumented at routine level through HySpec. The experiments have been performed over an architecture equipped with 1.5 GHz Itanium 2 processor and Intel compiler icc v9.0, using libpfm to measure the execution speed.

5.1 Applying Specialization on ATLAS

ATLAS (Automatically Tuned Linear Algebra Software) is a set of programs which acquires adaptive approach to tune itself at compile time for different parameters such as cache and memory sizes. We took into consideration two versions of ATLAS-3.6.0 library code: the "reference code", which is optimized regardless of platform and the "fully specialized and tuned code" which is optimized and tuned with hardware parameters.

Note: In the remaining text, we will refer ATLAS* to be the version of reference code and ATLAS to be tuned/optimized code and invoked by ATLAS as default.

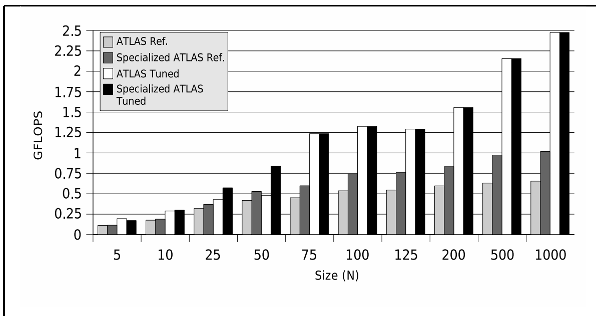

Figure 7. BLAS-I(DAXPY)

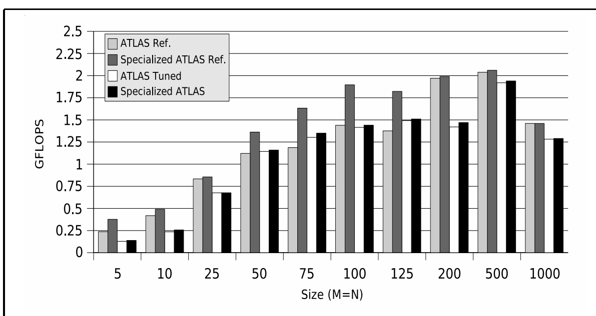

Figure 8. BLAS-II(DGEMV)

[3] The combination is selected for smallest overhead

5.1.1 BLAS-1

For BLAS-1(vector-vector operations), the performance results of the routine DAXPY have been shown in Figure 7. For ATLAS*, the *ATL_drefaxpy* routine has been specialized with *incX* and *incY* parameters. Due to the availability of value, there is a decrease of 30 % in object code size. Moreover, the pipelining and unrolling factor differed in the specialized and unspecialized versions but it is also dependent on the value for which the code is specialized.

For ATLAS code, the template for dynamic specialization found with interval of smaller values shows small improvement in performance. The parameter N is specialized for the routine *ATL_daxpy_xp1yp1aXbX* to obtain the templates. Despite the low-overhead specialization, speedup is not large since execution time of the specialized routine is very small as compared to overall execution time. Similarly, for other invocations (larger values of N), the code optimizations resembled with those performed for standard version.

5.1.2 BLAS-2

For ATLAS* BLAS-2 (matrix-vector operation), hybrid specialization is able to achieve speedup for some cases as shown in Figure 8. The routine DGEMV is specialized with the values of M and N (dimensions of input matrix). The speedup is achieved mainly through dynamically specialized templates (most of them requiring only 2 instructions to be specialized during execution). For original code, the compiler versioned the loops at the object code level, however with specialized code, these versions were removed and there was 70% decrement of code size. Moreover, the number of prefetch instructions was also reduced.

For ATLAS DGEMV, the speedup is not large since specialized (with M and N) code differed slightly (only in code schedule) from that of the standard version.

5.1.3 BLAS-3

Similarly for BLAS-3 (matrix-matrix operations), performance results of two operations DGEMM and DSYMM have been shown in Figure 9.

For ATLAS* DGEMM, the main speedup comes from the template that is generated for smaller values. The parameters M and K are specialized for the routine *ATL_drefgemmTT*. The binary template used for these values contained 2 instructions to be specialized, thereby causing a very small overhead of runtime specialization. In this case, the compiler was able to reduce the number of prefetches together with a better code schedule. With larger values of M and K, other versions of template code and static versions could produce a very small speed-up since

the specialized code resembled with that of unspecialized code.

For ATLAS DGEMM, the routine *ATL_dJIK0x0x0TT6x1x1_aX_bX* is specialized with values of *lda, ldb* and *K* (dimensions of input, output and intermediate result resp.), and the routine *ATL_mmJIK* is specialized for different values of parameter *K*. The execution results have been given in Figure 9. There is no significant speedup since ATLAS DGEMM code is already specialized for different sizes based on cache parameters. Moreover, all the loops have already been unrolled in the source code depending upon matrix sizes thereby making it difficult for any high-level code transformation to produce effective optimized code. In specialized version, the code size was relatively decreased together (with different number of stages of software pipeline) than that in the original code.

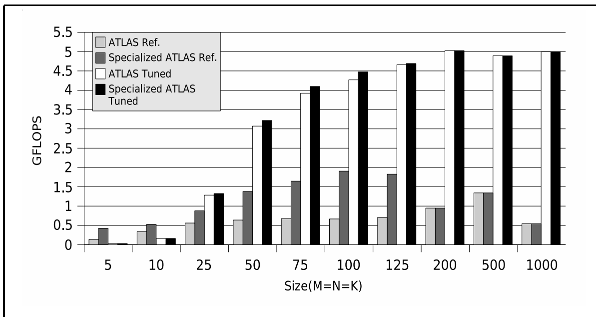

Figure 9. BLAS-III(DGEMM)

5.2 FFTW Library

FFTW is a library which contains routines written in C to compute Discrete Fourier Transform (DFT) of real and complex data and of arbitrary input size in *O(nlogn)*. It searches for best optimal codelets for computation of DFT. FFTW *wisdom* can be generated to reduce search time during execution.

For our experiments, the library FFTW-3.0.1-fma was optimized for complex numbers incorporating the *wisdom* generated in FFTW_EXHAUSTIVE mode. The code has been compiled with -O3 together with other default compilation options. Figure 10 shows the results for complex DFTs of powers of 2 obtained after hybrid specialization of FFTW codelets. Multiple codelets are invoked for computation of single DFT. In our case, the *standard* codelets for the powers of 2 were specialized. These codelets required variant number of instructions in the templates to be specialized for different intervals. It is clear that the main potential to achieve speedup through specialization exists for the smaller and the larger values of DFT size. For medium range values, where large codelets are selected

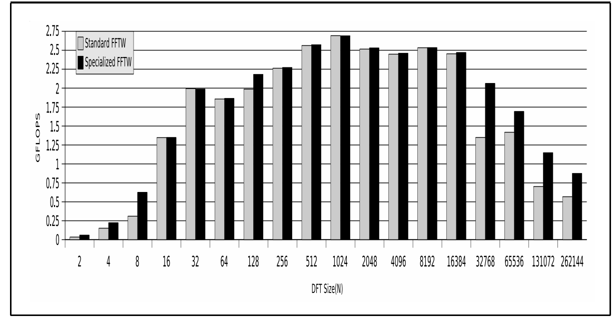

Figure 10. FFTW 3.0

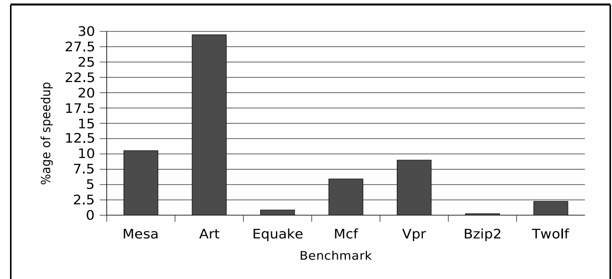

Figure 11. SPEC Benchmarks

by FFTW Wisdom, the number of calls to the specialized codelet is too small to impact the overall application execution time by a large factor.

5.3 SPEC Benchmarks

Different benchmarks in CPU2000 suite have been optimized using hybrid specialization with reference inputs. The speedup %age obtained w.r.t standard code has been shown in Figure 11.

For benchmark *art*, the *train_match* function is specialized. A good speedup is obtained as compiler was able to find the loop bounds thereby reducing the number of target instructions with a better schedule. In *mesa* benchmark, the function *sample_linear_1d* has been specialized with the loop counter. Both the codes were inlined, however the availability of the parameter resulted in more aggressive transformations than those for the original code. For *equake*, the *smvp* function is specialized, which did not produce any big difference due to the specializing parameter. Moreover, the overhead of specialization was also a factor to not produce good speedup. This is different in the case of *vpr* where the runtime specializer is invoked very rare since the parameter value remains the same for multiple invocations. The function *route_net* has been specialized for *vpr*. In case of *mcf*, the specialization of function *primal_bea_mpp* resulted in less number of prefetches. The compiler was able to adequately unroll the code since the parameter was used as stride. For *bzip2* benchmark, the

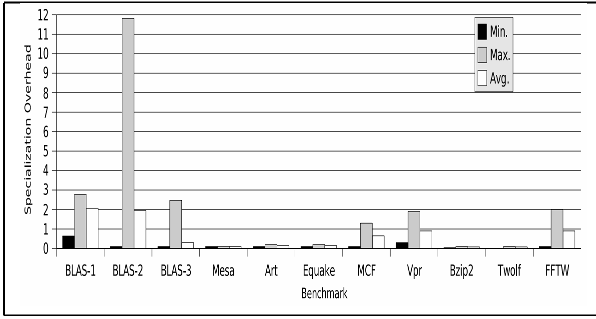

Figure 12. Specialization Overhead

function *hbAssignCodes* is specialized. However, the candidate code is very small and the large frequency of change in specializing parameter's value was the main barrier in achieving good speedup. In *twolf* benchmark, the function *term_newpos_a* is specialized. The specialized code contained slightly less memory accesses to produce a small speedup for the application.

6 Overhead and Size Concerns

A summarized view of overhead with respect to application execution time is also shown in Figure 12. It is to be noted that overhead is very small for execution of application function with large code. This is due to the fact that the overhead of dynamic specialization is limited (12 to 20 CPU Cycles per generated instruction). Moreover, we specialize a fixed number of binary instructions, which is very small as compared to entire code size.

In addition to performance gain, dynamic part of hybrid specialization serves effectively to reduce the code explosion caused by static specialization. With all the versions(including templates) and dynamic specializers, the code size increase is almost 9% for ATLAS and 6% for FFTW and less than 2% for each of SPEC benchmarks.

The specializer itself is very small (4.8 Kbytes). For each code dynamically specialized less than 20 binary instructions (4 bytes long on Itanium) need to be specialized. The maximum time needed to specialize one function is 400 CPU Cycles (20 instructions * 20 cycles).

7 Related Work

Tempo [7, 22] is a specializer that can perform compile-time and runtime specialization. The static compile-time activity by Tempo includes partial evaluation that is only applicable when the values are static (i.e. already known). In hybrid specialization, we expose the unknown values to generate template. Therefore the template is more specialized in our case than the one generated through Tempo specializer. In normal programs where most of the values are dynamic, the overhead of specialization will increase to a huge factor thus degrading the performance of the application. However, in the hybrid specialization approach, only calculation of specializing values is required since we are specializing a small number of binary instructions. Moreover, runtime activities other than optimizations, such as code buffer allocation and copy, incur large amount of overhead thereby making it suitable for code to be called multiple times.

C-Mix [20] is a specializer that is able to perform partial evaluation only at static compile-time. It can specialize the code by propagating information up to branches and then generating different specialized constructs. However it works when information is available at static compile-time, which is not the case for most of the real-life applications.

Tick C('C)[23] is a superset of ANSI C and uses ICODE and VCODE interfaces together with *lcc* retargetable intermediate representation to generate dynamic code. It is able to achieve large speedup by performing optimizations during execution of code. However, its code generation activity incurs a large amount of overhead requiring more than 100 calls to amortize the overhead. In case of hybrid specialization approach, we minimize the runtime overhead with generation of optimized templates at static compile time. Some other dynamic code generation systems have been suggested in [15, 18, 17] 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. Similarly, DCG [10] and other approaches in [12, 11, 6] suggest efficient dynamic code generation and specialization systems, however these systems are different in that these can not be used to produce *generic* templates requiring large number of dynamic template versions for each different specializing value.

Some optimization frameworks such as ADORE [19], Dynamo [1], Strata [13] and DAISY [9] have incorporated continuous monitoring to optimize/transform binary code at run-time. However, our framework specializes a small number of instructions relying heavily on offline profiles. Through this approach, we are able to keep the runtime overhead minimum but at the cost of reduced adaptability for largely varying execution behaviour of the application.

8 Conclusion and Future Work

This article presents a method combining static and dynamic specialization (through versioning of some parameters and a lightweight dynamic specializer), relying on the quality of the code generated at static compile time. Both approaches collaborate to give good speedups compared to the original code, even for highly optimized code such as

FFTW, ATLAS and SPEC benchmarks. This approach is not based upon selection of either static or dynamic specialization by making any cost-effective comparison, rather it combines both of them to an extent sufficient enough to obtain speedup. However, we are dependent on the partial evaluation or other optimizations performed by the compiler.

An important aspect of this automated hybrid specialization approach is the minimized overhead of runtime specialization. Most of the specialization work is performed at static-compile time and template generation includes a minimal generalization so as to make it valid for a large set of values. Since, the template is generated at static compile time, it is highly optimized, and during execution, only a small number of binary instructions is then specialized.

Moreover, this approach provides a solution to the code expansion issue caused by versioning and constitutes an interesting alternative to usual tradeoff between library size and performance achieved.

As future work, we intend to embed the implementation framework of hybrid specialization into XLanguage [8] compiler. Moreover, the static compile-time analysis is being extended to automatically search for candidate parameteres to specialize and also take into account smaller parts of program code instead of complete function code.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[2] M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P. Knijnenburg, M. O'Boyle, E. Rohou, R. Sakellariou, H. Shepers, A. Seznec, E. A. Sthr, and H. A. G. Wijshoff. Oceans: Optimizing compilers for embedded applications. In *Proceeding of Euro-Par98*, 1998.

[3] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, 1997.

[4] B. R. Childers, J. W. Davidson, and M. L. soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *NSF Workshop on Next Generation Software*, April 2003.

[5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. *In ACM Symposium on Principles of Programming Languages, Pages 493-501, 1993*, 1993.

[6] C. Consel, L. Hornof, François Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation. International Seminar.*, pages 54–72, Dagstuhl Castle, Germany, 12-16 1996. Springer-Verlag, Berlin, Germany.

[7] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3es), 1998.

[8] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the comParallel Architectures and Compilation Techniques representation of multiple program versions. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, LNCS, Hawthorne, New York, Oct. 2005. Springer-Verlag.

[9] K. E. and E. R. Altman. Daisy: dynamic compilation for 100 In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 26–37, New York, NY, USA, 1997. ACM Press.

[10] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 263–272, San Jose, California, 1994.

[11] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 163–178. ACM, June 1997.

[12] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc : An expressive annotation-directed dynamic compiler for c. Technical report, Department of Computer Science and Engineering,University of Washington, 1999.

[13] S. K., B. R. Childers, J. W. Davidson, and M. L. soffa. Overhead reduction techniques for software dynamic translation. In *In Proceedings of 18th International Parallel and Distributed Processing Symposium*, April 2004.

[14] L.Djoudi, D.Barthou, P.Carribault, C.Lemuet, J.-T. Acquaviva, and W.Jalby. Exploring application performance: a new tool for a static/dynamic approach. In *Proceedings of the 6th LACSI Symposium*, Santa Fe, NM, Oct. 2005.

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

[16] M. Leone and P. Lee. Optimizing ml with run-time code generation. Technical report, School of Computer Science, Carnegie Mellon University, 1995.

[17] M. Leone and P. Lee. A Declarative Approach to Run-Time Code Generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[18] M. Leone and P. Lee. Dynamic Specialization in the Fabius System. *ACM Computing Surveys*, 30(3es), 1998.

[19] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. *Journal of Instruction-Level Parallelism*, 6, April 2004.

[20] H. Makholm. Specializing c- an introduction to the principles behind c-mix. Technical report, Computer Science Department, University of Copenhagen, June 1999.

[21] S. Meloan. The java hotspot performance engine: An in-depth look. Technical report, Sun Microsystems, 1999.

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

[23] M. Poletto, W. C. Hsieh, D. R. Engler, and F. M. Kaashoek. 'c and tcc : A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21:324–369, March 1999.