Automated Empirical Tuning of a Multiresolution Analysis Kernel[*]

Haihang You[†] Keith Seymour[†] Jack Dongarra[‡] Shirley Moore[†]

November 10, 2006

1 Introduction

As CPU speeds double every couple of years following Moore's law[1], memory speed lags behind. Because of this increasing gap between the speeds of processors and memory, in order to achieve high performance on modern systems new techniques such as longer pipeline, deeper memory hierarchy, and hyper threading have been introduced into the hardware design. Meanwhile, compiler optimization techniques have been developed to transform programs written in high-level languages to run efficiently on modern architectures[2, 3]. These program transformations include loop blocking[4, 5], loop unrolling[2], loop permutation, fusion and distribution[6, 7]. To select optimal parameters such as block size, unrolling factor, and loop order, most compilers compute these values with analytical models referred to as model-driven optimization. In contrast, empirical optimization techniques generate a large number of code variants with different parameter values for an algorithm, for example matrix multiplication. All these candidates run on the target machine, and the one that gives the best performance is picked. With this empirical optimization approach ATLAS[8, 9], PHiPAC[10], OSKI[11], and FFTW[12] successfully generate highly optimized libraries for dense and sparse linear algebra kernels and FFT respectively. It has been shown that empirical optimization is more effective than model-driven optimization[13].

2 Generic Code Optimization (GCO) Framework

Current empirical optimization techniques such as ATLAS and FFTW can achieve good performance because the algorithms to be optimized are known ahead of time. We are addressing this limitation by extending the techniques used in ATLAS to the optimization of arbitrary code. Since the algorithm to be optimized is not known in advance, compiler technology is required to analyze the source code and generate the candidate implementations. The ROSE project[14, 15] from Lawrence Livermore National Laboratory provides, among other things, a source-to-source code transformation tool that can produce blocked

[*]This work was supported by the Department of Energy SciDAC program under grant no. DE-FC02-06ER25761.

[†]Department of Computer Science, University of Tennessee, Knoxville

[‡]Department of Computer Science, University of Tennessee, Knoxville and Oak Ridge National Laboratory

and unrolled versions of the input code. Combined with our search heuristic we can use the ROSE LoopProcessor to perform empirical code optimization [16, 17]. For example, we can direct the LoopProcessor to perform automatic loop blocking at varying sizes, which we can then evaluate to find the best block size for that loop. To perform the evaluations, we have developed a test infrastructure that automatically generates a timing driver for the routine based on a simple description of the arguments. Since the search space may be too large to feasibly perform an exhaustive search, we have implemented search strategies based on established optimization techniques, such as the simplex method and genetic algorithms. We have also implemented a random search to determine whether there is any benefit to using these techniques or if it will suffice to just choose a certain number of points at random.

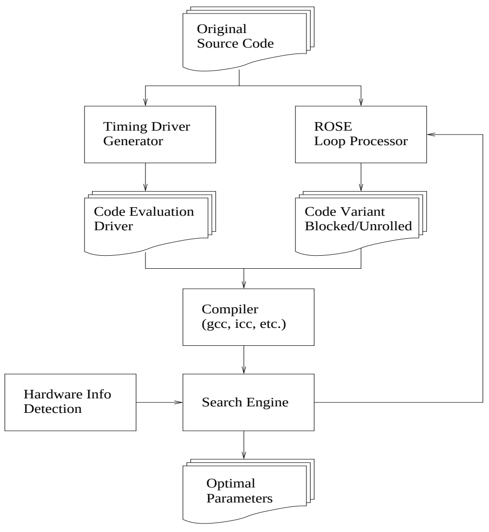

Figure 1: GCO Framework

Figure 1 shows the overall structure of the Generic Code Optimization system. The code is fed into the loop processor for optimization and separately fed into the timing driver generator which generates the code that actually runs the optimized code variant to determine its execution time. The results of the timing are fed back into the search engine. Based on these results, the search engine may adjust the parameters used to generate the next code variant. In the future, the initial set of parameters could be estimated based on the characteristics of the hardware (e.g. cache sizes).

3 Tuning the Multiresolution Analysis Kernel with GCO

As part of our participation in the Performance Engineering Research Institute (PERI)[*], we were made aware of an opportunity to apply our GCO framework to a performance limiting kernel for the MADNESS framework for adaptive multiresolution methods in multiwavelet bases [†] [18]. The kernel is encapsulated in the function `doitgen` in the original source code.

[*] http://www.peri-scidac.org/

[†] http://www.csm.ornl.gov/ccsg/html/projects/madness.html

Feature	Pentium 4	Woodcrest	Opteron
Processor Speed	1.7GHz	3.0 GHz	2.0GHz
L1 Instruction	12KB	32KB	64KB
L1 Data	8KB	32KB	64KB
L2	256KB	128KB	1024KB
OS	Linux	Linux	Linux
Compiler	gcc 3.4.4	gcc 3.4.6	gcc 3.4.6

Table 1: Processor Specifications

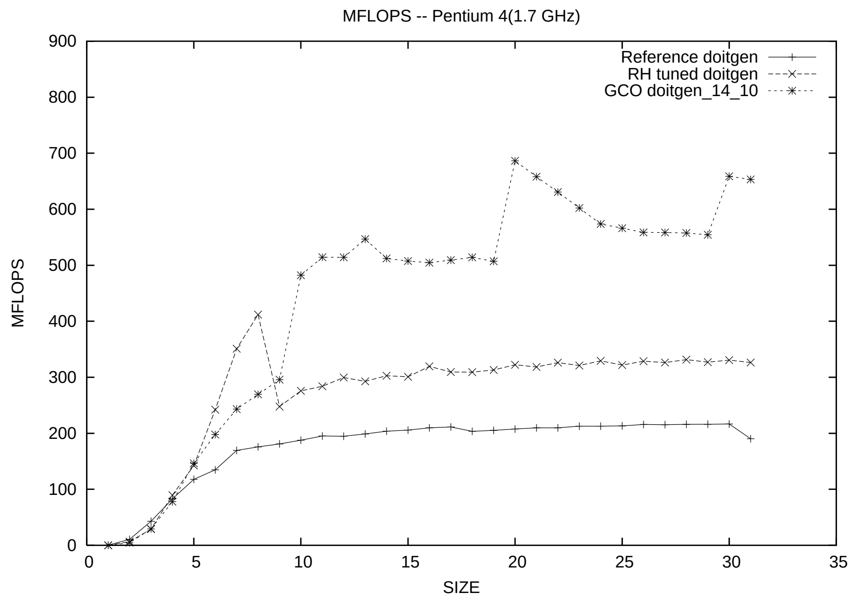

Figure 2: Pentium 4

The function `doitgen` is a good candidate for empirical tuning since it has a loop nest that can be optimized and the loops contain no external function calls. Since the ROSE LoopProcessor does not yet support Fortran, we translated the code to C using `f2c`. Section 4.1 shows the reference code after the conversion. In Section 4.3, the special comments at the top of the wrapper code are directives used by the GCO system to generate the appropriate testing driver and Makefile. These argument specification directives are written by hand. The function `doitgen` has a relatively small search space because the upper bound of each dimension of the input array is 31. Therefore, the more sophisticated search techniques were not necessary and a brute force search was used. We tested all block sizes from 1 to 31 and for unrolling, we limited the maximum unroll amount to the selected block size. Otherwise, if the unroll amount is greater than the block size, the unrolled section would not be executed. So far we have done experiments on Intel Pentium 4, Intel Core Duo (Woodcrest), and AMD Opteron. The specifications of these platforms are shown in Table 1. Section 4.2 shows the GCO-generated code on Opteron. A comparison of the performance of the reference code, the hand-tuned code, and the GCO generated `doitgen` code is shown

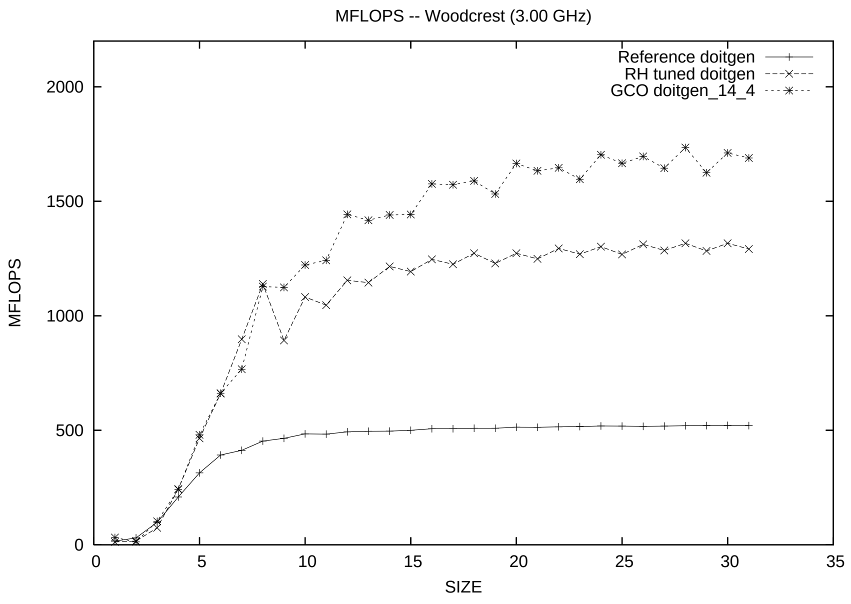

Figure 3: Woodcrest

in Figures 2, 3, and 4. The hand-tuned code (designated "RH tuned doitgen" in the graphs) was unrolled to varying depths by hand. The GCO-generated code is about 1.5 to 2 times faster than the hand-tuned version, and 2 to 3 times faster than the reference code. With the GCO approach, after writing the argument specification directives, the tuning process is automatic and took less than five minutes in this case.

4 Source Code

4.1 Reference doitgen

```
int doitgen_ref__(double *a, int *ia1, int *ia2,                    doitgen_ref__
      int *ia3, double *x, int *ldx, int *np,
      int *nq, int *nr, int *ns)
{
  /* System generated locals */
  int x_dim1, x_offset, i__1, i__2, i__3, i__4;

  /* Local variables */
  static int p, q, r__, s;
  static double t0[30];
  static int qr, pqr;
  static double sum;
  static int sqr;

  /* This is the original version ... a new port */
```

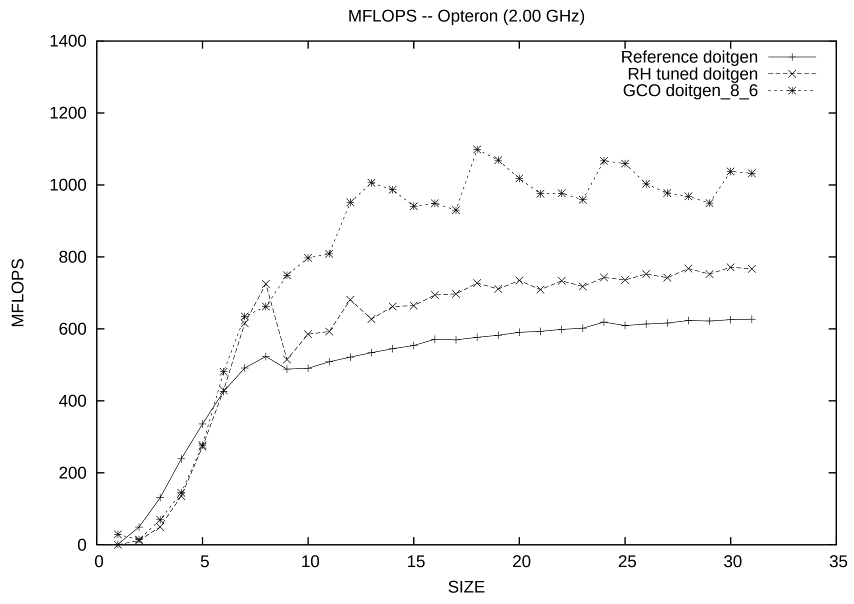

Figure 4: Opteron

```
/* to another processor could start from this. */

/* Parameter adjustments */
 --a;
 x_dim1 = *ldx;
 x_offset = 1 + x_dim1;
 x -= x_offset;

/* Function Body */
 i__1 = *nr;
 for (r__ = 1; r__ <= i__1; ++r__) {
   i__2 = *nq;
   for (q = 1; q <= i__2; ++q) {
     qr = (q - 1) * *ia2 + 1 + (r__ - 1) * *ia3;
     i__3 = *np;
     for (p = 1; p <= i__3; ++p) {
       sum = 0.;
       sqr = qr;
       i__4 = *ns;
       for (s = 1; s <= i__4; ++s) {
         sum += a[sqr] * x[s + p * x_dim1];
         sqr += *ia1;
       }
       t0[p - 1] = sum;
     }
```

```
      pqr = qr;
      i__3 = *np;
      for (p = 1; p <= i__3; ++p) {
        a[pqr] = t0[p - 1];
        pqr += *ia1;
      }
    }
  }
  return 0;
} /* doitgen_ref__ */
```

4.2 GCO-generated doitgen

```
int min(int ,int );

int doitgen_ref__(double *a,int *ia1,int *ia2,int *ia3, double *x,
      int *ldx, int *np, int *nq, int *nr, int *ns)
{
  int _var_0;
/* System generated locals */
  int x_dim1;
  int x_offset;
  int i__1;
  int i__2;
  int i__3;
  int i__4;
/* Local variables */
  int p;
  int q;
  int r__;
  int s;
  double t0[30];
  int qr;
  int pqr;
  double sum;
  int sqr;

/* Parameter adjustments */
  --a;
  x_dim1 = *ldx;
  x_offset = 1 + x_dim1;
  x -= x_offset;
/* Function Body */
  i__1 = *nr;
  for (_var_0 = 1; _var_0 <= i__1; _var_0 += 8) {
    for (r__ = _var_0; r__ <= min(i__1,(_var_0 + 7)); r__ += 1) {
```

doitgen_ref__

```
for (q = 1; q <= i__2; q += 1) {
  qr = ((q − 1) * *ia2 + 1) + (r__ − 1) * *ia3;
  i__3 = *np;
  for (p = 1; p <= i__3; p += 1) {
    sum = 0.0;
    sqr = qr;
    i__4 = *ns;
    for (s = 1; s <= −5 + i__4; s += 6) {
      sum += a[sqr] * x[(s + p * x_dim1)];
      sqr += *ia1;
      sum += a[sqr] * x[((1 + s) + p * x_dim1)];
      sqr += *ia1;
      sum += a[sqr] * x[((2 + s) + p * x_dim1)];
      sqr += *ia1;
      sum += a[sqr] * x[((3 + s) + p * x_dim1)];
      sqr += *ia1;
      sum += a[sqr] * x[((4 + s) + p * x_dim1)];
      sqr += *ia1;
      sum += a[sqr] * x[((5 + s) + p * x_dim1)];
      sqr += *ia1;
    }
    for (; s <= i__4; s += 1) {
      sum += a[sqr] * x[(s + p * x_dim1)];
      sqr += *ia1;
    }
    t0[(p − 1)] = sum;
  }
  pqr = qr;
  i__3 = *np;
  for (p = 1; p <= −5 + i__3; p += 6) {
    a[pqr] = t0[(p − 1)];
    pqr += *ia1;
    a[pqr] = t0[((1 + p) − 1)];
    pqr += *ia1;
    a[pqr] = t0[((2 + p) − 1)];
    pqr += *ia1;
    a[pqr] = t0[((3 + p) − 1)];
    pqr += *ia1;
    a[pqr] = t0[((4 + p) − 1)];
    pqr += *ia1;
    a[pqr] = t0[((5 + p) − 1)];
    pqr += *ia1;
  }
  for (; p <= i__3; p += 1) {
    a[pqr] = t0[(p − 1)];
    pqr += *ia1;
```

```
        }
      }
      i__2 = *nq;
    }
  }
  return 0;
/* doitgen_ref__ */
```

4.3 Wrapper doitgen

```
#include "f2c.h"

/*$ATLAS ROUTINE DOITGEN_REF */
/*$ATLAS SIZE 1:31:1 */
/*$ATLAS ARG IA1          IN    int    1        */
/*$ATLAS ARG IA2          IN    int    $size    */
/*$ATLAS ARG A[IA2][IA2][IA2] INOUT double $rand */
/*$ATLAS ARG X[IA2][IA2]  IN    double $rand    */
/*$ATLAS ARG LDX          IN    int    $size    */
/*$ATLAS ARG NP           IN    int    $size    */
/*$ATLAS ARG NQ           IN    int    $size    */
/*$ATLAS ARG NR           IN    int    $size    */
/*$ATLAS ARG NS           IN    int    $size    */

extern int doitgen_ref__(double *a, int *ia1, int *ia2,                doitgen_ref__
        int *ia3, double *x, int *ldx, int *np, int *nq,
        int *nr, int *ns);

/* Subroutine */ int doitgen_ref(int ia1, int ia2,
        double *a, double *x, int ldx, int np, int nq,
        int nr, int ns)
{
    int ia3=ia2*ia2, mu=1, xvt_dim1=ia2, xvt_dim2=ia2;
    double *xvt;
    int x_offset = 1 + xvt_dim1 * (1 + xvt_dim2);

    x -= x_offset;
    xvt=&x[(mu * xvt_dim2 + 1) * xvt_dim1 + 1];
    doitgen_ref__(a, &ia1, &ia2, &ia3, xvt, &ldx, &np, &nq, &nr, &ns);


    return 0;
} /* doitgen_ref_wrap */
```

5 Conclusion

We have demonstrated that the Generic Code Optimization system is effective at optimizing the `doitgen` computational kernel code. With less effort than it would take to tune by hand, we achieved better performance than both the hand-tuned version and the version generated by a general-purpose optimizing compiler. Also the resulting C code is readable enough that further optimizations not provided by the LoopProcessor could be performed by hand.

References

[1] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] David A. Padua and Michael Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[4] Qing Yi, Ken Kennedy, Haihang You, Keith Seymour, and Jack Dongarra. Automatic Blocking of QR and LU Factorizations for Locality. In *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, 2004.

[5] Robert Schreiber and Jack Dongarra. Automatic Blocking of Nested Loops. Technical Report CS-90-108, Knoxville, TN 37996, USA, 1990.

[6] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[7] Utpal Banerjee. A Theory of Loop Permutations. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 54–74. Pitman Publishing, 1990.

[8] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[9] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[10] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[11] Richard Vuduc, James Demmel, and Katherine Yelick. OSKI: A library of automatically tuend sparse matrix kernels. In *Proc. SciDAC 2005, Journal of Physics: Conference Series*, volume 16, San Francisco, CA, June 2005.

[12] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[13] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A Comparison of Empirical and Model-driven Optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 63–76. ACM Press, 2003.

[14] Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.

[15] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. Classification and untilization of abstractions for optimization. In *The First International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, Oct 2004.

[16] Haihang You, Keith Seymour, and Jack Dongarra. An effective empirical search method for automatic software tuning. Technical Report ICL-UT-05-02, Computer Science Department, University of Tennessee, May 2005.

[17] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. Technical Report CS-TR-2006-006, Computer Science Department, University of Texas at San Antonio, 2006.

[18] R.J. Harrison, I. Fann G, T. Yanai, and G. Beylkin. Multiresolution quantum chemistry in multiwavelet bases. In *Proc. International Conference on Computational Science (ICCS 2003)*, volume 2657-2660, pages 103–110, Melbourne, Australia, 2003. Springer-Verlag Lecture Notes in Computer Science.