# Code Crumpling: A Straight Technique to Improve Loop Performance on Cache Based Systems

Stephan Seidl

Center for High-Performance Computing (ZHR)
Dresden University of Technology
D-01062 Dresden, Germany
seidl@zhr.tu-dresden.de

**Abstract.** While parallelization has made important progress during the last few years, the sequential performance relatively stagnates, although we have taken the 1 GHz hurdle. Moreover, we have to put tens of processors into action to be competitive with vector machines. Once an application is analyzed by means of performance tools like Vampir [3], the user is typically faced with some loops which consume most of the CPU time. To improve loop performance, the present paper assumes that it is just enough in many cases to apply a generalized blocking technique with respect to the size of one primary cache line, or two of them occasionally. Its primary goal is to maximize the number of user operations divided by the expected value for the number of drawn primary cache lines. Another experience is that the inner loop body has to be well-stuffed and has to have as little memory write accesses as possible. More than 50 Fortran/C example loops have been investigated so far, and essential results are presented. The performance in speed correlates with results which we have seen for routines in the BLAS libraries on different platforms. Finally, because of its clear goals, this procedure can also easily be taught.

## 1 Introduction

The present investigations were mainly motivated by some experimental results from [5]. As an example, figures 1-4 show the matrix multiplication performance on T3E-600 and Origin2000. The different curves in each figure represent all the possible permutations with respect to loop nest level exchanges, i.e. they come from loop interchanges. In case that the output matrix is stored in transposed form, the results look similarly. At first we can see here that both machines run with more than half the peak performance when executing Fortran code. Furthermore, the left-sided speeds do not depend as strongly on cache size limitations as the right-sided ones do. Finally, loop interchanges do not change the results with Fortran on T3E-600. Putting all these facts together, a primal consequence is that high speeds are really possible on cache based systems with RISC processors, even if the caches are very small. Accordingly, loop-tuning transformations here should try to gain peak performance.

One more result comes from [4]. For table 1, the existence of an infinitely fast processor has been presupposed, which executes sign changes on floating-point

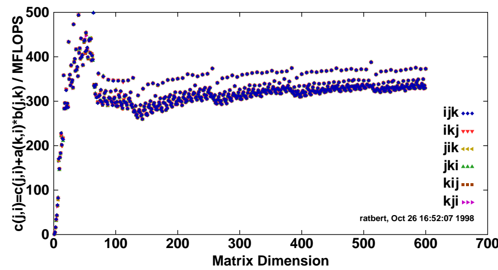


**Fig. 1.** Fortran-coded matrix multiplication on T3E-600 with streams off

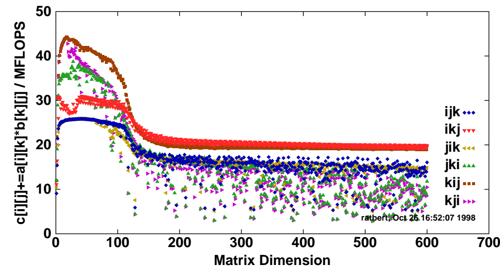


**Fig. 2.** C-coded matrix multiplication on T3E-600 with streams off

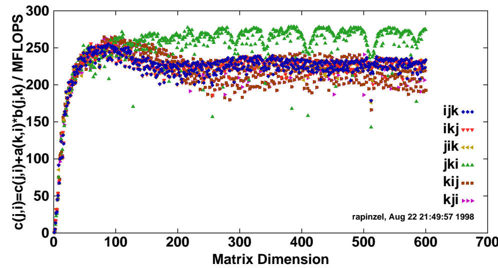


**Fig. 3.** Fortran-coded matrix multiplication on 195-MHz Origin2000

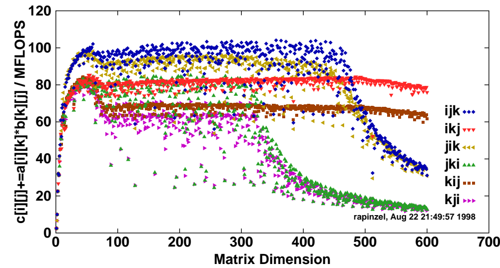


**Fig. 4.** C-coded matrix multiplication on 195-MHz Origin2000

**Table 1.** Sign change performance limits caused by the Origin2000 memory hierarchy level the data reside on

| Mnemonic | Level | MFLOPS |
|---|---|---|
| in-register | 0 | peak ? |
| on-chip cache | 1 | peak ? |
| SL cache | 2 | 75.5 |
| home memory | 3 | 20.3 |
| same corner | 4 | 14.5 |
| neighbor corner | 5 | 13.6 |
| next but one corner | 6 | 12.9 |
| spatial diagonal | 7 | 12.2 |

data without paying attention to the possibility of accelerating this task by means of suitable interventions, i.e. the processor does nothing but sit and wait for the arrival of incoming data. For peak performance, table 1 suggests that we have to maximize the primary on-chip cache hit rate mainly, whereas a big SL cache only increases the *ground speed*.

To do so, there are a lot of possible transformations. They can be found in [2] and [6], for example. The problem remaining for the user is to figure out which of them are the right ones. To predict the result of a particular loop transformation with respect to its cache behavior, the present paper uses a merit function described in the following section.

## 2 On the Merit Function

The scalar merit function $f$ is defined by the number of inner loop user operations divided by the expected value for the number of drawn primary cache lines due to the access to all operands there. $f$ is proportional to the speed, seen from the point of view of the memory accesses, i.e. it is proportional to the really observed speed as long as we are far from the processor's peak performance. One knows that the latter condition is very often fulfilled on cache based RISC systems. On the other hand, of course, near the peak performance, the real speed goes into saturation, so that a simple correlation with $f$ cannot exist any longer.

```
  for (i = 0; i < n; i += 1) {
  for (j = 0; j < n; j += 1) {
  for (k = 0; k < n; k += 1) {
  c[i][j] += a[i][k] * b[k][j]; /* u1=2 */
/* w1=0       w2=1/m     w3=1 */
  } } }
```
(1)

Illustratively, (1) shows the initial situation for the matrix multiplication. The number of user operations inside the inner loop is $u_1 = 2$. The probabilities for drawing a new primary cache line are $w_1 = 0$, $w_2 = 1/m$, and $w_3 = 1$, respectively. $m$ should be the number of floating-point operands that a primary cache line can hold which typically equals 4. Furthermore, $n \rightarrow \infty$ has been assumed. Hence, $f$ can be obtained to $f = f_0 = u_1/(w_1 + w_2 + w_3) = 2m/(m + 1)$. (2) would be the result after inner loop unrolling, where $n$ is supposed to be even.

```
  for (i = 0; i < n; i += 1) {
  for (j = 0; j < n; j += 1) {
  for (k = 0; k < n; k += 2) {
  c[i][j] += a[i][k + 0] * b[k + 0][j]; /* u1=2 */
/* w1=0          w2=1/m        w3=1 */
  c[i][j] += a[i][k + 1] * b[k + 1][j]; /* u2=2 */
/* w4=0          w5=1/m        w6=1 */
  } } }
```
(2)

Since (2) yields the same value for $f$ as (1) does, inner loop unrolling is of no effect here, at least with respect to the access patterns. These things change with (3) being the $j$-loop unrolling case.

```
  for (i = 0; i < n; i += 1) {
  for (j = 0; j < n; j += 2) {
  for (k = 0; k < n; k += 1) {
  c[i][j + 0] += a[i][k] * b[k][j + 0]; /* u1=2 */
   /* w1=0       w2=1/m        w3=1   */
  c[i][j + 1] += a[i][k] * b[k][j + 1]; /* u2=2 */
   /* w4=0       w5=0          w6=1/m */
  } } }
```
(3)

$a[i][k]$ is reused ($w_5 = 0$), and the expensive access to $b[k][j + 0]$ ($w_3 = 1$) is followed by a unit-stride one ($w_6 = 1/m$). In fact, $w_6$ differs from 0, even $w_3$ equals 1. As shown in figure 5, this comes from the statistical treatment of several alignment situations. $w_3$, $w_6$, $w_{3\nu}$, and $w_{6\nu}$ are the appropriate probabilities.

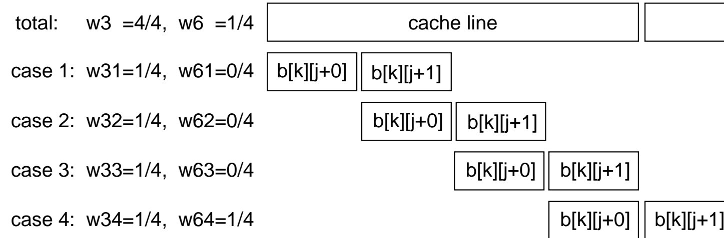


**Fig. 5.** Treatment of alignment situations in (3)

As a result, one gets $f = f_2 = 2qm/(m+q)$, assuming that $q$ is the unroll factor. For $q > 1$, $g_2 = f_2/f_0 = q(m+1)/(m+q)$ is always greater than 1, hence we should have an observable speedup from now on. Finally, the outer loop is unrolled.

```
for (i = 0; i < n; i += 2) {
for (j = 0; j < n; j += 2) {
for (k = 0; k < n; k += 1) {
c[i + 0][j + 0] += a[i + 0][k] * b[k][j + 0]; /* u1=2 */
   /* w1=0            w2=1/m        w3=1   */
c[i + 0][j + 1] += a[i + 0][k] * b[k][j + 1]; /* u2=2 */
   /* w4=0            w5=0          w6=1/m */
c[i + 1][j + 0] += a[i + 1][k] * b[k][j + 0]; /* u3=2 */
   /* w7=0            w8=1/m        w9=0   */
c[i + 1][j + 1] += a[i + 1][k] * b[k][j + 1]; /* u4=2 */
   /* wa=0            wb=0          wc=0   */
} } }
```
(4)

(4) goes on to improve in performance according to $f = f_3 = 2q^2m/(m+2q-1)$ and $g_3 = f_3/f_0 = q^2(m+1)/(m+2q-1)$. For example, with $q = 4$ and $m = 4$, the predicted speedup is $g_3 = 7.27$ compared with $g_2 = 2.50$ for (3).

Summarizing here, in particular cases, the defined merit function can easily be determined. Since most of the existing computers are cache based systems, this function helps to estimate the effect of the different loop transformations described in literature.

## 3 The Favored Sequence of Code Transformations

[6] is very helpful for a brief description of the loop code transformation sequence favored here. Of course, [6] must be read from the point of view of scalar optimization. As a result, one should get completely re-rolled code with a small number of fat loops, whereby inner loops should have unit-stride as well as non-unit-stride accesses. After that, unrolling is applied which is controlled by the value of the function $f$ from above. Finally, all the non-scalar accumulators, if any, should explicitly be replaced by scalar ones, and, perhaps, other scalar optimization should follow.

## 4 Two Examples

The inner loop of the first example from real life shows (5). This example is a level-2 type problem, combining a vector and a triangular matrix. Identifiers which begin with `j` are integers, and the ones beginning with `p` are pointers. `f`, as the first letter, denotes floating-point data. One gets $f = f_{10} = 2m/(m+1)$.

```
for (j0 = 0; j0 < j1; j0++)
  f0 += *(p0 + j0) * *(p1 + ((j2 - j0) * j0 >> 1));
```
(5)

(6) shows the well-stuffed inner loop after an optimization. As in (5), one has unit-stride vector accesses, whereas the triangular matrix is accessed group-wise now. $f$ is obtained to $f = f_{11} = 2qm/(m+q)$, as for (3). Assuming 4 floating-point numbers per cache line again, (6) should be $\frac{10}{3}$ times faster than (5). Moreover, (6) should work at $\frac{2}{3}$ of the theoretical speed which one would get for $q \to \infty$.

```
for (j3 = 0; j3 < j4; j3++) {
  f0 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 0);
  f1 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 1);
  f2 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 2);
  f3 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 3);
  f4 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 4);
  f5 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 5);
  f6 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 6);
  f7 += *(p2 + j3) * *(p3 + ((j5 - j3) * j3 >> 1) + 7);
  }
```
(6)

All the memory accesses are read-only and the accumulations can happen in registers. Therefore, a compiler should have plenty room for optimization. Figures 6 and 7 represent the appropriate results. The *Fortran* and the *C* case are based on (5), their performances equal on both machines. The *Advanced* case here means C language according to (6). Unfortunately, like the second one, too, this example does not have a corresponding BLAS implementation. While T3E-600 follows acceptably the theory, the acceleration on Origin2000 is too small, mainly caused by its big SL caches, i.e. the multiply used vector does not get lost completely because of inner loop activities. Finally it should be denoted that the calculations to one curve point started always with empty caches.

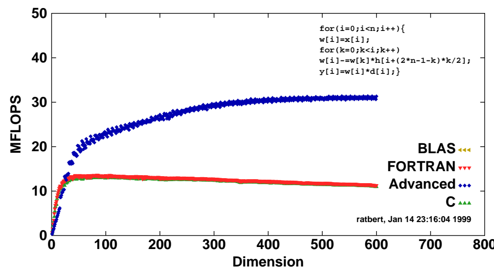


**Fig. 6.** Level-2 type performance on T3E-600 with streams off

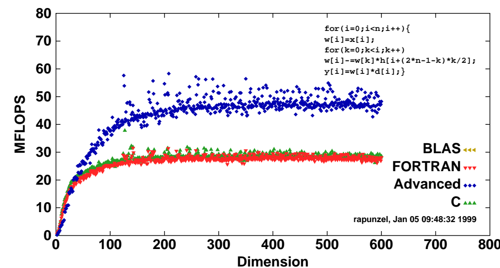


**Fig. 7.** Level-2 type performance on 195-MHz Origin2000

The second example is a level-3 type problem. Its inner loop looks exactly like the inner loop (5), although it combines a rectangular and a triangular matrix. (7) is the result of an optimization. $n$ is one of the problem dimensions, with $n \gg 1$. The rectangular matrix is accessed by way of four unit-stride *memory streams*. The triangular matrix access group size was chosen to be half of the one of (6), because each of the group members is used here four times. Basically, it is much easier to optimize a level-3 type problem than a lower type one, on the other hand, if we have a level-3 type subproblem in our algorithm, then we should like to see peak performance there.

```
for (j6 = 0; j6 < j7; j6++) {
  f0 += *(p4 + n * 0 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 0);
  f1 += *(p4 + n * 1 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 0);
  f2 += *(p4 + n * 2 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 0);
  f3 += *(p4 + n * 3 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 0);
  f4 += *(p4 + n * 0 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 1);
  f5 += *(p4 + n * 1 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 1);
  f6 += *(p4 + n * 2 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 1);
  f7 += *(p4 + n * 3 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 1);
  f8 += *(p4 + n * 0 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 2);
  f9 += *(p4 + n * 1 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 2);
  fa += *(p4 + n * 2 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 2);
  fb += *(p4 + n * 3 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 2);
  fc += *(p4 + n * 0 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 3);
  fd += *(p4 + n * 1 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 3);
  fe += *(p4 + n * 2 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 3);
  ff += *(p4 + n * 3 + j6) * *(p5 + ((j8 - j6) * j6 >> 1) + 3);
  }
```
(7)

(7) yields $f = f_{21} = 2q^2m/(m + 2q - 1)$, accidentally, this is the same value as for (4). Again, if we assume 4 floating-point numbers per cache line, (7) should be $\frac{80}{11} \approx 7.27$ times faster than (5). This value cannot significantly be improved, although the theoretical acceleration is unlimited with $q \to \infty$. For example, an acceleration of 16.84, i.e. an unroll factor $q = 8$ would need $8^2 = 64$ floating-point registers here. Figures 8 and 9 show the appropriate measurements which are satisfying at all.

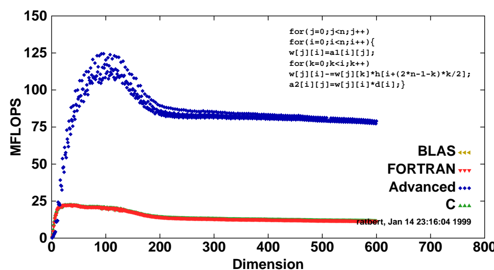


**Fig. 8.** Level-3 type performance on T3E-600 with streams off

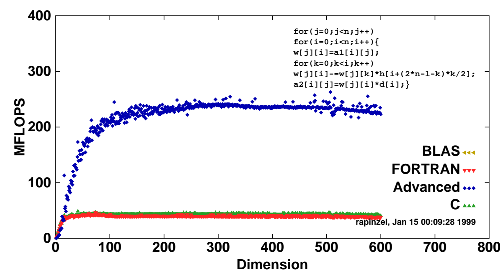


**Fig. 9.** Level-3 type performance on 195-MHz Origin2000

## 5 Conclusion

Once detected, the effect of hot-spots consisting of level-2 or higher level loop nests can be reduced by replacing these pieces of code by highly optimized and, at the same time, portable sequences, even if there are no appropriate BLAS implementations. Not only during tuning loop controlled operations, the behavior of cache based systems can be modeled with the help of the described merit function above. The theoretical and experimental results with hand-tuned codes are in agreement with [1, page 76]. Because of the small block size which is typically used here (*micro-blocking*), accesses to indirectly referenced arrays do not produce fundamental problems; one does not need more than a certain amount of additional integer registers. Carefully selected unroll levels do allow the compiler to save some registers for its own purposes. Interestingly, but not shown here, codes like the described ones also yield good speedups on new Intel processor based machines, although those CPUs have a very small register set. On the other hand, more than 500 MFLOPS have been observed on a 200-MHz POWER-3 workstation. Finally, it seems that there is no way of accelerating level-1 type loops without special hardware facilities, e.g. on the one hand the automatically but only upwards working T3E Streams and on the other hand the non-blocking prefetch instructions of the MIPS processor being more flexible but needing initiation by the user. And, of course, optimizing strategies for cache based systems differ completely from these for vector computers.

## Acknowledgments

I want to thank Prof. Dr. W.E. Nagel, Dresden University of Technology, and Dr. W. Oed, Cray Research, for helpful discussions in understanding many details of recent and gone hardware. Thanks are also due to Dr. J. Heinke, Dr. S. Maletti, and J. Weller at Computer Center of Dresden University of Technology for support on T3E and Origin2000.

## References

1. Dongarra J.J., Duff I.S., Sorensen D.C., and van der Vorst H.A.: *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, 1998, ISBN 0-89871-428-1.
2. Dowd K., Severance C.R.: *High Performance Computing, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, 1998, ISBN 1-56592-312-X.
3. Nagel W.E., Arnold A., Weber M., Hoppe H-C., and Solchenbach, K.: *VAMPIR: Visualization and Analysis of MPI Resources*. Supercomputer 63, Vol. 12, No. 1, 1996, pp. 69-80.
4. Seidl S.: *Access Delays Related to the Main Memory Hierarchy on SGI Origin2000*. http://armoise.saclay.cea.fr/~workshop/Documents/Final_Papers/Stephan_Seidl_11_Perf_opt_1.ps.
5. Seidl S., Nagel W.E.: *CRAY T3E and SGI Origin2000: Merging Architectures from the User's Point of View*. Proc. of the Fourth European SGI/Cray MPP Workshop (Sep 10-11, 1998, IPP, Garching, Germany), H. Lederer and F. Hertweck, eds., IPP R/46, 1998, pp. 6-19.
6. Zima H., Chapman B.: *Supercompilers for Parallel Computers*. ACM Press, 1990, ISBN 0-201-17560-6.

Originally published under http://www.cineca.it/mpp-workshop/proceedings.htm, 1999.
Also available as internal report ZHR-IR-9913.