CPU performance tuning based on the type of application

version 1.1

Operations and Computer Technologies Division
RIKEN Center for Computational Science
Dec. 1, 2021

Revision History

Version	Content revised	Date
1.1	Initial English version	12/1/2021

Terminology

In some inserted pictures, the following words in the right column are used as the same meaning as the left one in the main text.

main text	included pictures
sequential	consecutive
rate	ratio

Contents

Chapter 1

Classifying applications for performance analysis

1.1 Classifying applications by bytes-to-flops (B/F) ratio

From the perspective of single central processing unit (single-CPU) performance, applications can be broadly subdivided into two categories. The first category consists of applications for which the number of bytes transferred in and out of memory is relatively small compared to the number of floating-point arithmetic operations (flops) performed; such applications are said to have a *low B/F (bytes-to-flops) ratio*. An example of an operation with a low B/F ratio is $N \times N$ matrix-matrix multiplication. Here, the number of numerical values that must be transferred in and out of memory scales like N^2 (the number of entries in each matrix), while the number of arithmetic operations scales like N^3, so the ratio of (memory transfers)/(floating-point operations) scales—in principle—like $1/N$. As memory-transfer volumes are typically expressed in terms of byte counts, each double-precision arithmetic memory transfer involves 8 bytes, thereby yielding a B/F ratio of $8/N$. Thus, for this class of application, the B/F ratio diminishes as N increases.

In matrix-matrix multiplication and other procedures with low B/F-ratios, each numerical value deposited in cache memory is reused multiple times for arithmetic operations, thus allowing calculations to achieve high single-CPU performance. In other words, low-B/F-ratio applications can be thought of as applications that execute "in cache."

The second category consists of applications with high B/F ratios—that is, applications requiring a large number of memory transfers per floating-point operation. Applications of this type are fundamentally limited in their ability to use cache memory efficiently, and thus generally cannot achieve high single-CPU performance. In other words, high-B/F-ratio applications can be thought of as executing "in memory," rather than in cache.

An example of a task in this category is matrix-vector multiplication, for which the number of memory transfers required per arithmetic operation is roughly 1/2 in principle. Assuming 8 bytes per transfer for double-precision arithmetic, this yields a B/F ratio of $8/2 = 4$, which is significantly larger than that required for matrix-matrix multiplication.

1.2 A performance model accounting for memory, L2 cache, and functional unit

Let's consider a program being executed on a system with given main memory, L2 cache, and floating-point arithmetic unit. Denote the data-transfer volume and effective bandwidth of main memory by M_{data} and M_{band}, the data-transfer volume and effective bandwidth of the L2 cache by $L2_{data}$ and $L2_{band}$, the number of arithmetic operations executed by the program by N_c, and the effective performance of the functional unit by P_{peak}. Next, let t_M denote the time required to execute the program assuming main memory to be the primary performance bottleneck. Similarly, let t_{L2} and t_C denote the program execution times while assuming the L2 cache and functional unit to be the performance-limiting components. Then, we have

$$t_M = M_{data}/M_{band} \tag{1.1}$$

$$t_{L2} = L2_{data}/L2_{band} \tag{1.2}$$

$$t_C = N_c/P_{peak}. \tag{1.3}$$

The program execution time t_E and peak performance ratio C_p are then given by

$$t_E = \max\{t_M, t_{L2}, t_C\} \tag{1.4}$$

$$C_p = N_c/(\max\{t_M, t_{L2}, t_C\} \times P_{peak}) \tag{1.5}$$

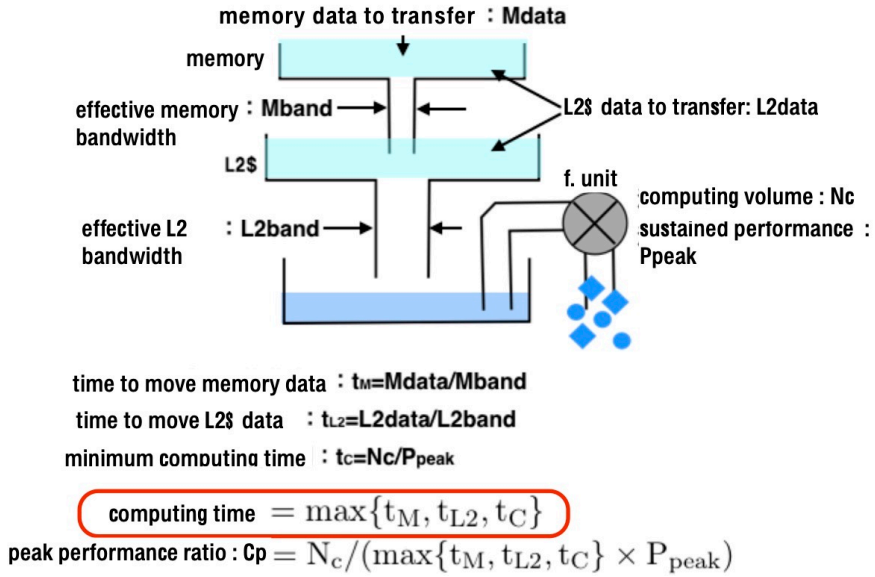


Figure 1.1: Performance model accounting for memory, L2 cache, and functional unit.

1.3 CPU performance-analysis reports: Busy times

FUGAKU generates CPU performance-analysis reports similar to that shown in Figure 1.2. The busy times quoted in these reports for memory, L2 cache, and functional unit agree approximately with the quantities t_M, t_{L2}, and t_C discussed in Section 1.2, while the program execution time quoted in the report agrees approximately with the quantity t_E. The reason for the approximate, not exact, agreement of these values is discussed below.

1.4 Application subcategories for busy-time analysis

As noted above, for single-CPU performance analysis, it is useful to classify applications into two broad categories. On the other hand, for *busy-time* analysis, it is useful to classify applications into *five* categories, as we will discuss in this section.

1.4.1 Optimally-tuned in-memory applications

The first category of applications for busy-time analysis consists of optimally-tuned in-memory applications. For applications of this type, the program execution time ideally matches the memory busy time as determined by the effective memory bandwidth, thus indicating that the program is taking full advantage of the effective memory bandwidth—or, equivalently, that the L2 cache and/or functional unit have excess capacity.

This situation is shown in Figure 1.3.

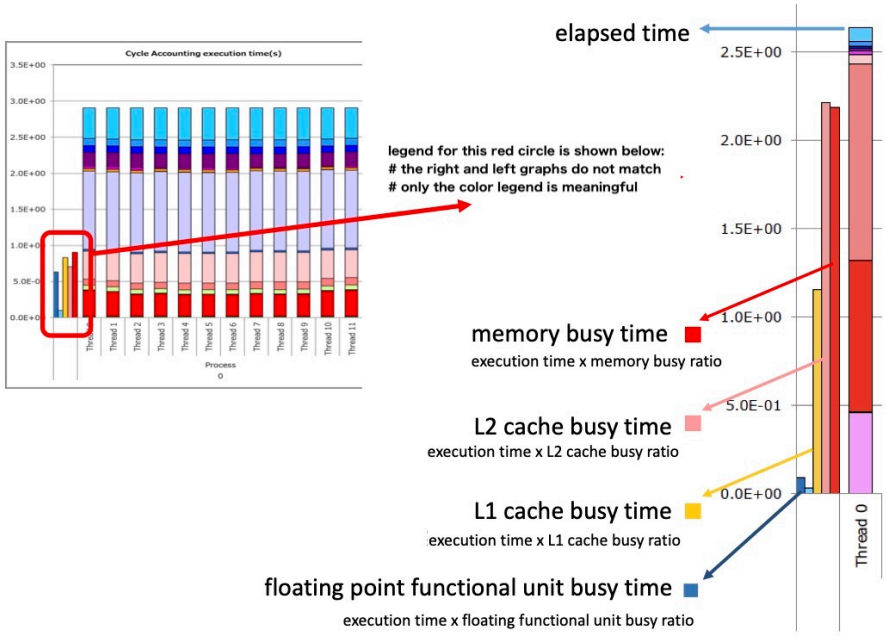


Figure 1.2: Fugaku-generated report of CPU performance and busy times.

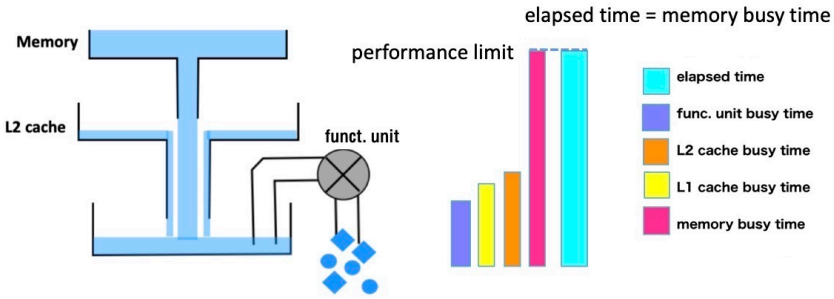


Figure 1.3: An optimally tuned in-memory application.

1.4.2 Optimally-tuned in-L2-cache applications (1)

The second category of applications for busy-time analysis consists of optimally-tuned in-L2-cache applications that are not functional unit bound. For applications of this type, the program execution time ideally matches the L2-cache busy time as determined by the effective L2-cache bandwidth, thus indicating that the program is taking full advantage of the effective L2-cache bandwidth—or, equivalently, that main memory and/or the functional unit have excess capacity.

This situation is shown in Figure 1.4.

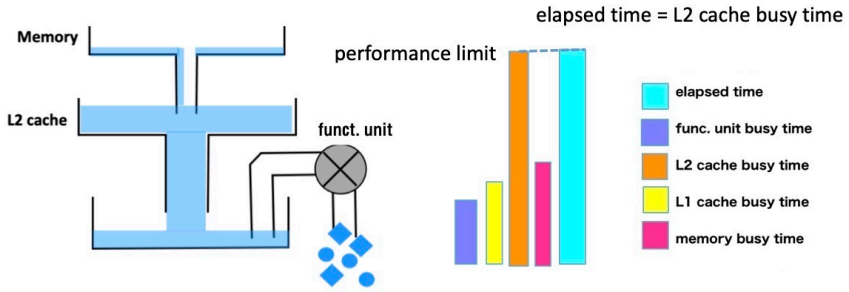


Figure 1.4: An optimally-tuned in-L2-cache application (1)

1.4.3 Optimally-tuned in-L2-cache applications (2)

The third category of applications for busy-time analysis consists of optimally-tuned in-L2-cache applications that are also functional unit bound. For applications of this type, the program execution time ideally matches the functional unit busy time as determined by the effective functional unit performance, indicating that the program is extracting the maximum possible performance available from the functional unit—or, equivalently, that main memory and/or the L2 cache have excess capacity.

This situation is shown in Figure 1.5.

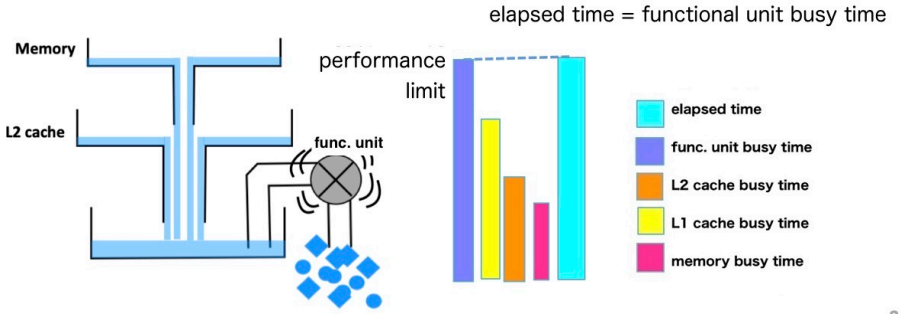


Figure 1.5: An optimally-tuned in-L2-cache application (2)

1.4.4 Optimally-tuned in-L1-cache applications

The fourth category of applications for busy-time analysis consists of optimally-tuned in-L1-cache applications. For applications of this type, the program execution time ideally matches the L1D-cache busy time, thus indicating that the program is taking full advantage of the available performance of the L1D cache. The busy time of the L1D cache is not determined solely by the effective bandwidth of that cache, but rather depends heavily on factors other than bandwidths. Therefore, applications of this type cannot be described by the model of Section 1.2, which is based on effective bandwidths. However, for optimally-tuned applications, the program execution time will match the L1D-cache busy time, thus indicating that main memory, the L2 cache, and/or functional unit have excess capacity.

This situation is shown in Figure 1.6.

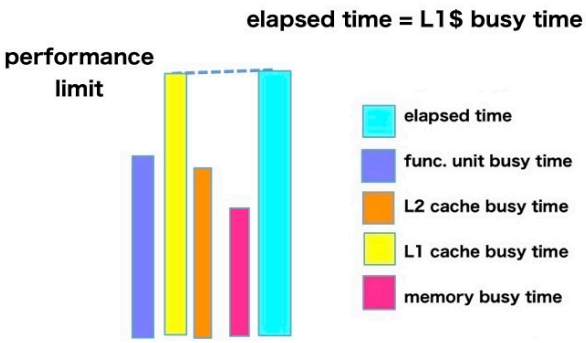


Figure 1.6: An optimally-tuned in-L1-cache application.

1.4.5 Insufficiently tuned in-memory or in-cache applications

The fifth and final category of applications for busy-time analysis consists of in-memory or in-cache applications that have not been tuned sufficiently thoroughly to yield optimal performance. For applications of this type, the program execution time exceeds the individual busy times of all hardware components, thereby indicating that the application fails to extract the maximum possible performance available from the hardware—or, equivalently, that all hardware components, including main memory, L2 cache, L1D cache, and functional unit, have excess capacity. This situation is shown in Figure 1.7.

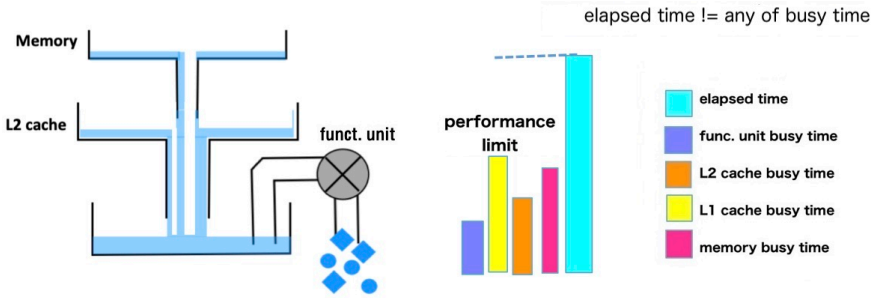


Figure 1.7: An insufficiently tuned in-memory or in-cache application.

1.5 Application subcategories for single-CPU performance analysis

In Section 1.1 we noted that, for analysis of single-CPU performance, applications might be broadly classified into two categories: those with low bytes-to-flops (B/F) ratios and those with high B/F ratios. Pursuing this line of thought slightly further yields the more granular classification scheme of Table 1.1, in which we distinguish six distinct application subcategories.

Subcategories 1 through 4 are subdivisions of the category of low-B/F-ratio applications. Two applications, both belonging to this broader class, can nonetheless exhibit significantly different performance levels depending on (a) whether or not the application can make use of the DGEMM library, and (b) whether or not the application performs manual cache blocking. Applications that perform manual cache-blocking can be further separated into those that use only the simplest data structures and loop structures versus those that employ slightly more complicated constructs—such as the list-indexed vectors commonly used in sparse linear algebra, in which the indices of array elements are themselves stored as components of integer-valued arrays (i.e., lists). The slight increase in the complexity of loop structures arising in such computations necessitates a different set of tuning techniques. (Meanwhile, applications involving *significantly* more complicated loop structures are generally unable to achieve high performance levels, even with tuning.) Taking these considerations into account yields the four distinct subcategories of low-B/F-ratio applications listed in rows 1-4 of Table 1.1.

Subcategory 1 consists of calculations that can be arranged in the form of $N \times N$ matrix-matrix multiplications. As noted in Section 1.1, for applications of this type, the numbers of memory transfers and

arithmetic operations scale respectively, like N^2 and N^3, thereby yielding low B/F ratios for moderate or large N. Common examples of calculations in this subcategory include *ab-initio* quantum-chemistry calculations via density-functional theory (DFT). Subcategory 2 includes applications in which low B/F ratios allow effective cache utilization and the bodies of computational loops are relatively short and simple. Examples include numerical differentiation via specialized finite-difference stencils.[1] Applications in this category generally achieve high performance, but unfortunately, practical examples are somewhat rare. Subcategory 3 consists of low-B/F-ratio calculations that cannot be implemented as matrix-matrix multiplications, but for which cache blocking is possible. Examples include computations of Coulomb interactions in molecular dynamics or gravitational interactions in gravitational many-body problems. Here, cache blocking is possible because we only need to load data for n particles into cache to facilitate the computation of n^2 particle-particle interactions. Subcategory 3 applications often use compound indices—consisting of lists of data fields—to label particles, which tends to yield programs in which the computational loop bodies[2] are somewhat complicated. The fourth subcategory consists of calculations with low B/F ratios but complicated loop bodies. Of the two types of numerical calculations arising in computational meteorology—namely, (a) numerical modeling of mechanical processes to predict the motion of fluids, and (b) numerical modeling of physical processes to predict the behavior of clouds and similar phenomena— type (b) calculations fall into this subcategory. Applications of this type carry out complicated sequences of arithmetic operations on relatively small amounts of data loaded from memory, and the programs thus generally run in-cache—but with loop bodies that tend to be long and convoluted. Another example of a computational task belonging to this category is the particle-in-cell[3] (PIC) method used in the modeling of plasmas and other systems.

In PIC calculations, the volume of mesh data needed to characterize a particle's local environment is small enough to reside in cache, but accessing individual particles often involves manipulating list-form data, thereby yielding programs with complicated structures and computational loops with long loop bodies. Thus, although the in-cache execution of subcategory-4 calculations suggests that applications in this class should be capable of achieving high performance, in practice, this expectation often remains unfulfilled due to the complexity of the programming required.

The final two rows of Table 1.1 are subdivisions of the broader class of applications with high B/F ratios. Applications with identical large B/F ratios can nonetheless exhibit significantly different performance levels depending on whether or not they require non-contiguous memory access via linked lists, and we define subcategories 5 and 6 based on this distinction.

Subcategory 5 consists of applications with high B/F ratios that do not involve indirect memory accesses. Tasks of this type commonly arise in calculations involving standard low-order stencils, and there are many examples of subcategory-5 applications, including fluid-dynamical modeling of mechanical processes in meteorology (mentioned above) and numerical modeling of seismic activity. Applications with high B/F ratios that *do* require indirect memory access comprise subcategory 6. Computations of this type frequently arise in engineering problems, such as fluid modeling and structural analysis via finite-element methods. Programs using indirect indexing generate random (non-sequential) memory accesses each time a list element is referenced, thus making these calculations unsuitable for execution on present-day scalar processor architectures.

In general, computational performance levels are highest for applications in subcategory 1 and deteriorate progressively as we proceed through subcategories 2, 3, 4, 5, and 6. (However, the order of subcategories 2 and 3 in this sequence can be reversed.)

1.6 Correspondences between application subcategories

In this section, we discuss correspondences between the two application-classification schemes discussed in the preceding sections.

The busy-time category of *optimally-tuned functional unit-bound in-cache applications* (Section 1.4.3) generally corresponds to single-CPU-performance subcategory 1: *applications that can be rewritten in the form of matrix-matrix multiplications*. Rewriting in this way and using a GEneral Matrix-to-matrix Multiply (GEMM) library custom-tuned for a given machine allows applications to attain the maximal computational performance available from the machine. Alternatively, some applications in this busy-time category correspond to single-CPU-performance subcategory 2: *applications with low B/F ratios and simple loop bodies*, for which we gave

[1] In finite-difference arithmetic, a *stencil* is a rule for computing the ith element of an output array as a weighted sum of elements $\{\cdots, i-1, i, i+1, \cdots\}$ in an input array.

[2] The *body* of a loop is the code executed on each loop iteration.

[3] A numerical modeling technique that tracks the behavior of particles in a computational lattice.

Table 1.1: Application subcategories for single-CPU performance analysis.

Subcategory	Description	Typical application
1	Expressible as matrix-matrix multiplication	First-principles quantum chemistry (density-functional theory)
2	Low-B/F ratio, simple loop bodies	High-order finite-difference stencils
3	Allows cache blocking	Molecular dynamics and gravitational many-body problems
4	Low B/F ratio, complex loop bodies	Plasma modeling; physical processes in meteorology; quantum chemistry
5	High B/F ratio, sequential and direct memory access	Mechanical processes in meteorology; fluid modeling; earthquake modeling; nuclear fusion
6	High B/F ratio with indirect memory access	Structural analysis or fluid modeling via finite-element methods

the example of high-order computational stencils. The performance attainable by applications of this type is generally not as impressive as what can be achieved with optimized GEMM libraries but can approach the maximal effective functional unit performance available in the absence of such libraries. One such example is discussed below.

Calculations falling within the busy-time categories of *optimally-tuned in-L2-cache applications* (Section 1.4.2) or *optimally-tuned in-L1D-cache applications* (Section 1.4.4) typically correspond to single-CPU-performance subcategory 3: *low-B/F-ratio calculations for which cache blocking is possible*, for which we gave the examples of molecular dynamics or gravitational many-body problems. Applications of this type generally cannot match the performance levels available for optimally-tuned functional unit-bound in-cache calculations but can achieve relatively high performance levels. Alternatively, some applications in these busy-time categories correspond to single-CPU-performance subcategory 2 (low B/F ratios with simple loop bodies).

The busy-time category of *suboptimally-tuned in-cache applications* (Section 1.4.5) typically corresponds to single-CPU-performance subcategory 4: *applications with low B/F ratios but complicated loop bodies*, for which we gave the examples of plasma modeling, modeling of physical processes in meteorology, and quantum-chemistry computations. The fact that these applications run in cache suggests that they should be able to achieve at least somewhat high performance, but in practice, the complicated nature of the programming involved tends to yield programs whose run times exceed all hardware busy times, i.e., programs that fail to achieve their highest possible performance levels. Some applications in this busy-time category can correspond more closely to CPU-performance subcategory 3, i.e., cache-blocking applications such as molecular-dynamics or gravitational many-body problems.

The busy-time category of *optimally-tuned in-memory applications* (Section 1.4.1) typically corresponds to single-CPU-performance subcategories 5 or 6: *applications with high B/F ratios* that do (subcategory 6) or do not (subcategory 5) require indirect memory access. Subcategory-5 calculations are typified by applications such as mechanical-process modeling in meteorology, fluid modeling, earthquake modeling, and nuclear fusion, while subcategory 6 includes applications such as structural analysis or fluid modeling via finite-element methods. Note that these categories are for applications that have been successfully tuned for optimal performance. In contrast, cases in which optimal tuning methods cannot be applied fall within the category of insufficiently-tuned in-memory applications, discussed in Section 1.4.5.

The correspondences discussed above are depicted graphically in Figure 1.8.

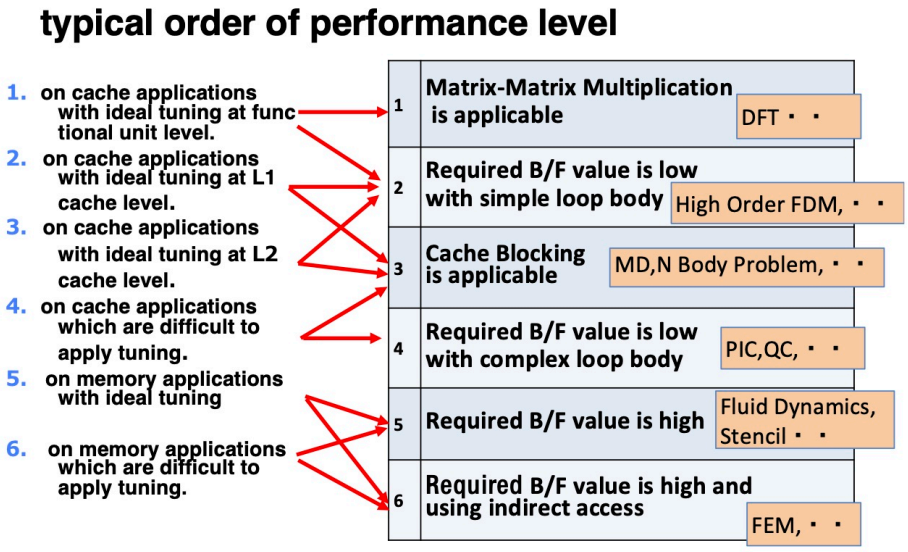


Figure 1.8: Correspondences between application categories for busy-time analysis (left) and single-CPU performance (right).

Chapter 2

What is single-CPU performance tuning?

In this chapter, we use the busy-time analysis of the previous chapter to answer the basic question: **What *is* single-CPU performance tuning?** In brief, single-CPU performance tuning can be defined as the task of ensuring that a program's run time agrees as closely as possible with the maximum busy time of any hardware component. This is done by using measured application performance and other data to classify the program's behavior—based on the single-CPU performance categories of Section 1.5, the busy-time categories of Section 1.4, and the correspondence between the two discussed in Section 1.6—and then evaluating whether or not the program's busy-time graph exhibits the expected form while addressing any problems that might arise. This general process is depicted schematically in Figure 2.1.

Figure 2.2 depicts a more involved example of program performance tuning consisting of a two-step process. In the first step, the program is rewritten to shift data accesses from main memory to the L2 cache. This reduces the main-memory busy time (magenta) in favor of L2-cache busy time (orange) and establishes a lower value for the theoretical minimum program run time (indicated by the maximum of the heights of the left four bars). In the second step, performance problems are addressed to reduce the actual program run time (cyan) to a value as close as possible to its theoretical limit.

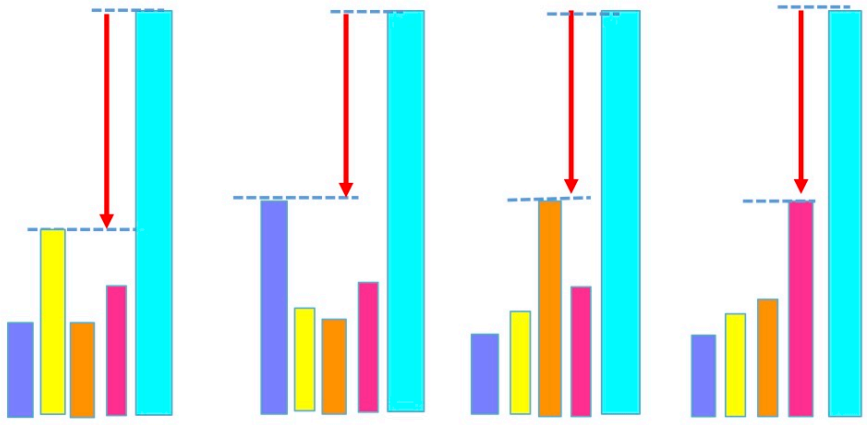


Figure 2.1: single CPU performance tuning (1)

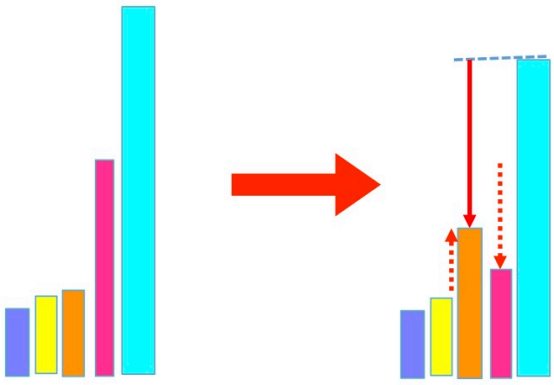


Figure 2.2: single CPU performance tuning (2)

Chapter 3

Case studies of busy-time analysis for various application types

In this section, we analyze busy times for specific examples of applications in the various categories of Section 1.4.

3.1 Memory-bound application

First, as an example of a memory-bound application, we consider busy times for an optimally tuned in-memory application. Our test program for this case is listed in Figure 3.1. A property of this test program is that it accesses main memory rather than running in cache. The use of `zfill` further ensures that storage operations do not generate transfers from memory to the L2 cache. In our runtime measurements, we run each program 10 times. For the code shown in Figure 3.1 the total number of inner-loop iterations is then $8000000 \times 24 \times 10 = 1.92 \cdot 10^9$. Each inner-loop iteration requires two double-precision values (16 bytes) to be transferred to/from memory, thereby yielding a total memory-transfer volume of $1.92 \cdot 10^9 \times 16 = 30.72$ GB. Assuming an effective memory bandwidth of 205 GB/sec (80% of peak) as measured for the STREAM triad, the quantity t_M defined in Section 1.2 (the program execution time assuming main memory to be the performance-limiting component) is $t_M = 30.72$ GB/(205 GB/s) = 0.15 s.

```
      integer,parameter :: M= 8000000
      integer,parameter :: N= 24
      real(8),intent(inout) :: c10(M+1,n)
      real(8),intent(in) :: c1(M+1,n)

        do j = 1,N
!OCL ZFILL
          do i = 1,M
           c10(i,j) = (z+c1(i,j)*x)* &
                                 c1(i,j)+z
          enddo
        enddo
```

Figure 3.1: Test program for which main memory is the performance-limiting component.

Figure 3.2 shows the CPU performance-analysis report generated upon execution of this program. Here, it should be noted that the actual measured run time is 0.15 seconds, in agreement with the estimate computed above. This indicates that the memory is achieving a bandwidth of roughly 205 GB/sec, on par with that of

the STREAM triad. The main-memory busy time reported in the CPU performance-analysis report should be 0.15 seconds, in agreement with the program execution time. However, from the figure, we see that, in fact, the reported busy times tend to fall slightly below this expected value. The reason for this is that CPU performance-analysis reports are computed using a normalization in which a busy ratio of 100% corresponds to the theoretical peak value. If we instead normalize so that 100% busy ratio corresponds to the *effective* peak value, the computed busy time agrees with the predicted value of t_M. Note that the L2-cache busy time stands out prominently. This is discussed in the following section.

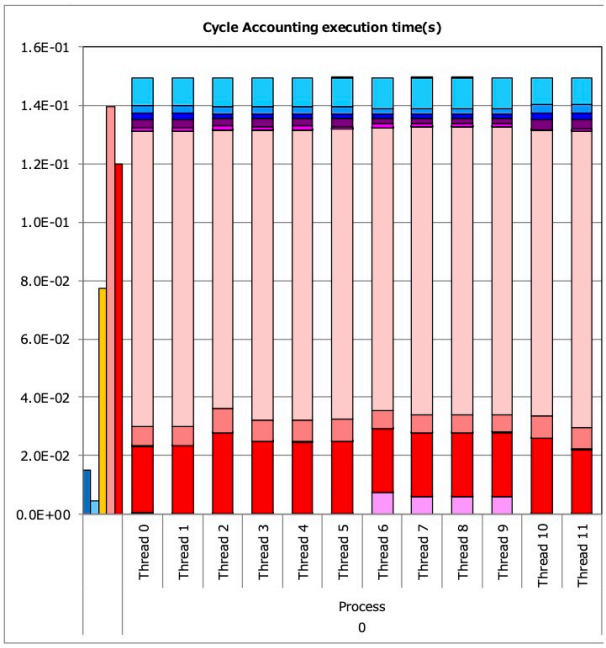


Figure 3.2: CPU performance-analysis report for memory-bound test program.

3.2 L2-cache-bound application

Next, as an example of an L2-cache-bound application, we consider an in-cache application for which optimal L2-cache tuning is possible, as discussed in Section 1.4. The code is listed in Figure 3.3. In this program, main-memory accesses and L2-cache accesses coexist, and `zfill` is not used, thus ensuring that storage operations generate data transfers from main memory to the L2 cache. In this example, each iteration of the innermost loop generates three main-memory accesses, two L2-cache accesses, and two functional unit operations, so we refer to it as a 3M-2L2-2F program. In our runtime measurements, we execute the program `N2` times, thus yielding a total of $3610 \times 60 \times 168 \times 60 = 2.18 \cdot 10^9$ innermost-loop iterations. Based on measurements performed separately, we use a value of 671 GB/sec (67.2% peak value) for the effective L2-cache bandwidth.

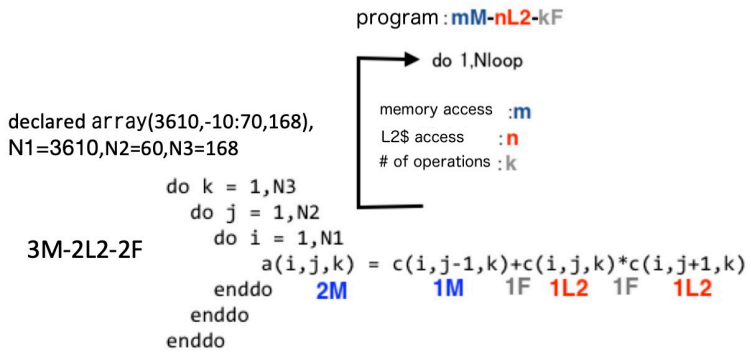


Figure 3.3: Test program accessing both main memory and the cache.

We can use the model of Section 1.2 to compute t_M for our 3M-2-2F program. With three double-precision numbers (24 bytes) transferred on each inner-loop iteration, the total data-transfer volume is $2.18 \cdot 10^9 \times 24 = 52.32$ GB. Therefore, for the memory-bound case, we estimate a runtime of $t_M = (52.32 \text{ GB})/(205 \text{ GB/s}) = 0.255$ seconds.

Next, we next compute t_{L2} for this program. On each inner-loop iteration, the program accesses two array elements (16 bytes) in the cache and three array elements (24 bytes) in main memory. The total volume of data transferred through the L2 cache is then $2.18 \cdot 10^9 \times (16 + 24) = 87.2$ GB, whereupon we estimate a value of $t_{L2} = (87.2 \text{ GB})/(671 \text{ GB/s}) = 0.13$ seconds for the runtime in the L2-cache-bound case. In the model discussed in Section 1.2, the program execution time t_E is greater than t_M and t_{L2}, which is t_M in this case.

Figure 3.4 shows the CPU performance-analysis report generated for our 3M-2L2-2F program. Here, it should be noted that the measured runtime of 0.255 seconds agrees with the estimate obtained above. The main-memory busy time reported in the performance analysis should be 0.255 seconds, in agreement with the program run time. However, in practice, it tends to fall short of this expectation, as shown in the graph. Also, the reported L2-cache busy time should agree with our estimated value of 0.13 seconds, but, in fact, it significantly exceeds this expectation. These findings are in agreement with the trends noted in the previous section.

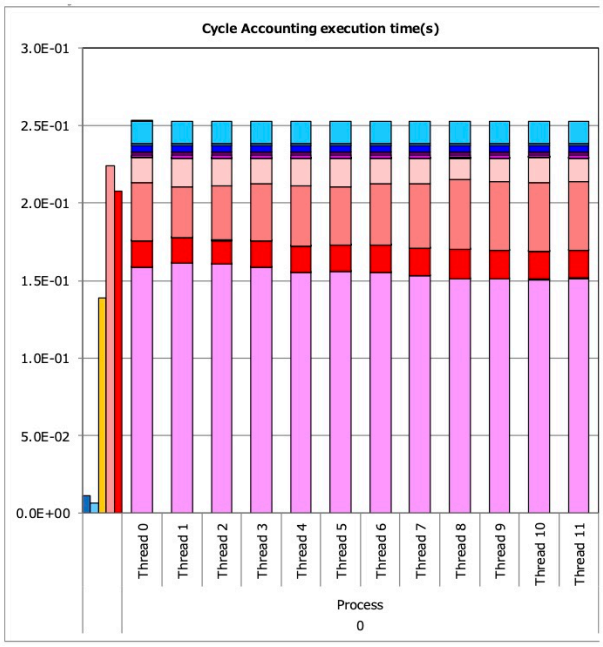


Figure 3.4: CPU performance-analysis report for the 3M-2L2-2F program.

Now, suppose we modify the program by increasing the number of arithmetic operations and L2-cache

accesses on each loop iteration, converting our 3M-2L2-2F program into a 3M-12L2-12F program. How do these modifications effect computational performance? As before, we estimate performance using the model of Section 1.2. Because the modified program involves the same number of memory accesses as the original program, we use the same estimate of 0.255 seconds for the execution time while assuming memory-bound conditions. Next, we compute t_{L2} for this program. We have 12 values (96 bytes) accessed from the L2 cache and three values (24 bytes) accessed from main memory, so the total volume of data transferred through the L2 cache is $2.18 \cdot 10^9 \times (96 + 24) = 261.6$ GB and the program execution time assuming L2-cache-bound conditions is $t_{L2} = (261.6 \text{ GB})/(671 \text{ GB/s}) = 0.39$ seconds. In the model discussed in Section 1.2, the program execution time t_E is the greater of t_M and t_{L2}, which is t_{L2} in this case. (Because, in this case, there are few arithmetic operations, the functional unit will not be the performance-limiting component. Therefore, we omit the t_C calculation.) Figure 3.5 shows the CPU performance-analysis report generated for our 3M-12L2-12F program. Here, it should be noted that the measured runtime of 0.384 seconds agrees with the estimate obtained above.

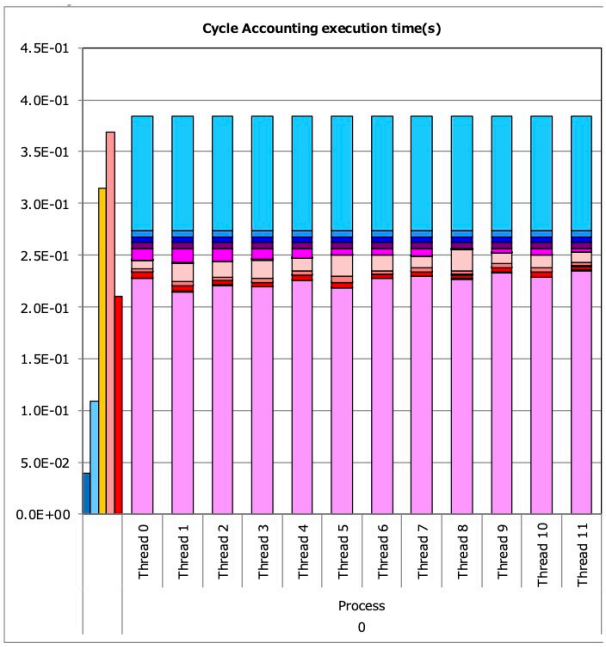


Figure 3.5: CPU performance-analysis report for 3M-12L2-12F program.

In this case, the execution time assuming L2-bound conditions (0.390 seconds) is in good agreement with the L2 busy time reported in the graph. This is in marked contrast to what we saw above for the 3M-2L2-2F program, in which the L2-bound estimate of execution time disagreed with the L2-cache busy time reported in performance charts. The reason for this is that the L2-cache busy ratio quoted in CPU performance-analysis reports is computed from L2-cache pipeline values, which are influenced by resources on the memory side when the memory busy ratio is high. More specifically, when the memory busy ratio is high, L2 prefetch retries are unavoidably included, thereby yielding high values for the apparent L2 busy ratio. In Section 3.1, we noted that CPU performance-analysis reports are computed using a normalization in which a busy ratio of 100% corresponds to the theoretical peak value and that the value of t_M defined in Section 1.2 is conceptually equivalent, but not identical, to the memory busy time. Similarly, for the reasons discussed above, the busy time of Section 1.3 is conceptually equivalent, but not identical, to the value of t_{L2} defined in Section 1.2.

3.3 functional unit bound application

Finally, as an example of an functional unit bound application, we consider an in-cache application for which optimal functional unit tuning is possible, as discussed in Section 1.4. First, we must determine how much performance can be extracted by increasing arithmetic operations with memory accesses held fixed. Test programs for this purpose are listed in Figures 3.6 and 3.7.

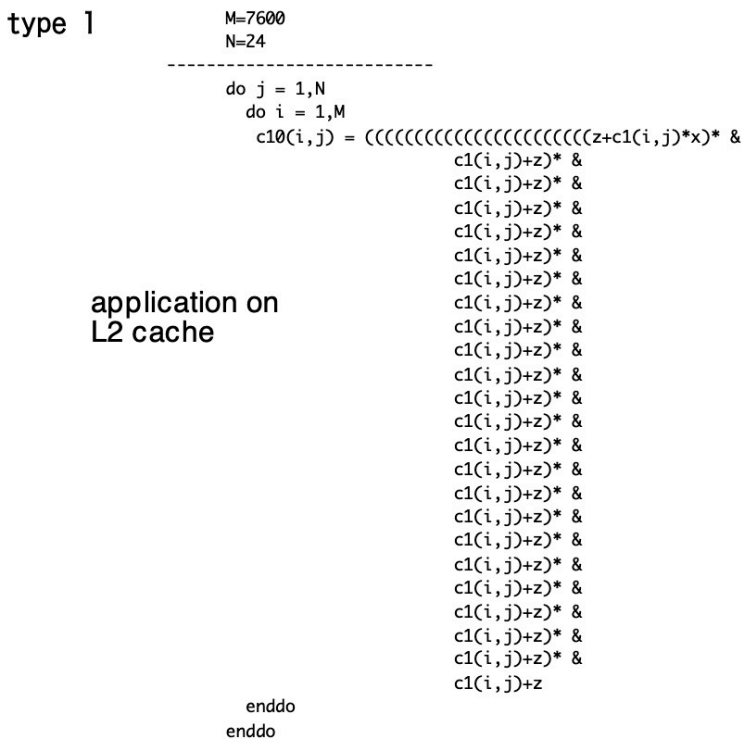


Figure 3.6: Program to test performance limits (K-computer version).

On the K computer, the use of type-1 coding achieves an arithmetic peak performance level of 88%, which is on par with the DGEMM library performance. In contrast, type-1 coding on Fugaku fails to achieve peak performance, while with type-2 coding, we achieve an arithmetic peak performance level of 68%. (On Fugaku, as on other machines, higher performance can be achieved by using DGEMM libraries or other codes that take full advantage of the L1D cache.) Type-1 coding involves deep chains of instructions, which are difficult to execute with high performance on Fugaku. Instead, high performance is achieved by using type-2 coding to sever instruction chains. Additionally, in comparison to the K computer, Fugaku has fewer registers, higher latency for arithmetic instructions, and larger single instruction, multiple data (SIMD) widths. These factors have the effect of slightly decreasing performance relative to arithmetic peak values.

As a specific example of an functional unit bound test program, we use the code listed in Figure 3.8. This is basically the same test program used in Section 3.2, modified to yield an application of type 3M-6L2-80F; we use this program to estimate execution times.

type 2

M=7600
N=24

```
do j = 1,N
   do i = 1,M
         ww1 = ((((((((((z+c1(i,j))* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z

         ww2 = ((((((((((x+c1(i,j))*x+ &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z)* &
                                c1(i,j)+z
         c10(i,j) = ww1 + ww2
       enddo
enddo
```

Figure 3.7: Program to test performance limits (Fugaku version).

```
integer,parameter :: N1= 3610
integer,parameter :: N2= 60
integer,parameter :: N3= 168

do k = 1,N3
   do j = 1,N2
     do i = 1,N1
        ww1 = (((((((c(i,j+1,k)+c(i,j,k)*c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))
        ww2 = (((((((c(i,j+1,k)*c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))* &
           c(i,j+1,k)+c(i,j-1,k))
        ww3 = (((((((c(i,j+2,k)*c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))
        ww4 = (((((((c(i,j,k)*c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+2,k)+c(i,j-2,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))
        ww5 = (((((((c(i,j+3,k)*c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))* &
           c(i,j+3,k)+c(i,j-3,k))
        a(i,j,k) = ww1 + ww2 + ww3 + ww4 + ww5
     enddo
   enddo
enddo
```

Theinnermost loop does:
- 3 memory access operations
- no use of ZFILL
- 6 L2 cache access operations
- 80 float. point operations

Figure 3.8: functional unit bound test program.

We run this program N2 times, so the number of inner-loop iterations is $3610\times60\times168\times60=2.18\cdot10^9$. We use the model discussed in Section 1.2 to compute t_M for this 3M-6L2-80F program: the total volume of data transfer to/from main memory is $2.18\cdot10^9\times$ (24 bytes)=52.32 GB, so the execution time assuming memory-bound conditions is t_M = (52.32 GB)/(205 GB/s) = 0.255 seconds. Next, we compute t_{L2} for this program. On each inner-loop iteration, the program accesses six values (48 bytes) in the L2 cache and three elements (24 bytes) in main memory; the total volume of data transferred through the L2 cache is then $2.18\cdot10^9\times(48+24)$ = 156.96 GB, whereupon we estimate a value of t_{L2} = (156.96 GB)/(671 GB/s) = 0.234 seconds for the runtime in the L2-cache-bound case. Finally, the total number of floating-point operations is $2.18\cdot10^9\times80 = 174.4\cdot10^9$ flops, so the estimated runtime for the functional unit bound case is $t_C = (174.4\cdot10^9 \text{ flops})/(768\cdot10^9 \text{ flops/s}\times0.68) = 0.334$ seconds.

In the model of Section 1.2, the program execution time t_E is the greatest of t_M, t_{L2}, and t_C, which is t_C in this case.

Figure 3.9 shows the CPU performance-analysis report generated for this 3M-6L2-80F program. Here, it

should be noted that the measured runtime of 0.331 seconds agrees with the estimate obtained above. The functional unit busy time reported in the performance analysis should be 0.331 seconds, in agreement with the execution-time estimate. However, it tends to be slightly lower. In analogy to the above, the busy time of Section 1.3 is conceptually equivalent, but not identical, to the value of t_C defined in Section 1.2.

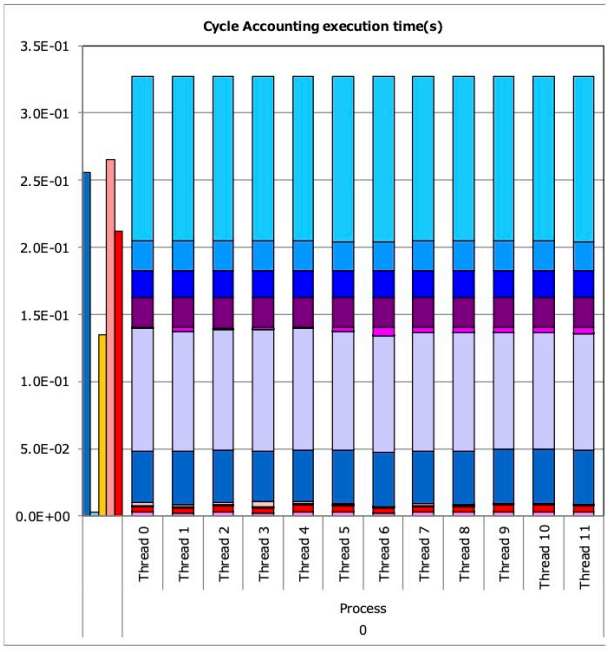


Figure 3.9: CPU performance-analysis report for 3M-6L2-80F program.

Chapter 4

Basic approaches for improving single-CPU performance on Fugaku

4.1 Basic strategies for improving CPU performance on Fugaku

As a prelude to discussing Fugaku performance tuning, we will first explain the software-pipelining (SWPL) techniques used for performance tuning on the K computer and the FX100. We will base our discussion on the simple computational code labeled "Target program" shown in Figure 4.1, which we assume to be running on a model machine with the hardware properties listed in the box at the upper left-hand side of the figure.

If we do not attempt to optimize instruction scheduling and simply compile and run the program in the bare-bones, unadorned form we see here, we obtain the sequence of machine instructions labeled "Original Loop" in the figure, whose execution proceeds as follows: First, array elements `B(1)` and `C(1)` are loaded from memory into registers. Three cycles later, the functional unit executes an `add` operation to compute their sum. Three cycles after that, the result is stored in memory as array element `A(1)`. This completes the processing required for the first element of the array, which evidently consumes seven cycles of CPU time. The procedure is then repeated for array elements 2, 3, $\cdots$, `n`, with each element consuming the same seven cycles of CPU time to process. Thus, in the absence of any instruction-scheduling optimizations, the full loop requires a total of $7n$ cycles to complete.

We achieve higher performance by reorganizing the sequence of machine instructions loaded into the pipelined configuration shown in Figure 4.1. In the pipelined version of the program, instructions to fetch array elements `B(i)` and `C(i)` from memory are issued on *every* CPU cycle (and not just every seventh cycle, as in the non-pipelined version). If, for example, the ith array elements are loaded from memory on CPU cycle p, then their sum is computed by the functional unit on cycle $p+3$ and stored in memory on cycle $p+6$. Thus, the processing of each array element completes on the seventh CPU cycle after it begins. This is as in the non-pipelined case, but with the difference that the 7-cycle processing intervals for distinct matrix elements now elapse *simultaneously* and in parallel, as opposed to the one-element-at-a-time structure of the non-pipelined case. By organizing—or *scheduling*—instructions in this way, the kernel portion of the sequence can generate results for one array element on each successive CPU cycle (as opposed to every seventh cycle in the non-pipelined case), whereupon the full loop for an n-element array requires a total execution time of just over n cycles. This is a significant improvement over the $7n$ cycles required in the non-pipelined case. Of course, in practice, there can be pre- and/or post-processing steps whose computational costs slightly reduce the magnitude of the speedup achieved by pipelining. However, the effect of such factors diminishes as the number of array elements increase.

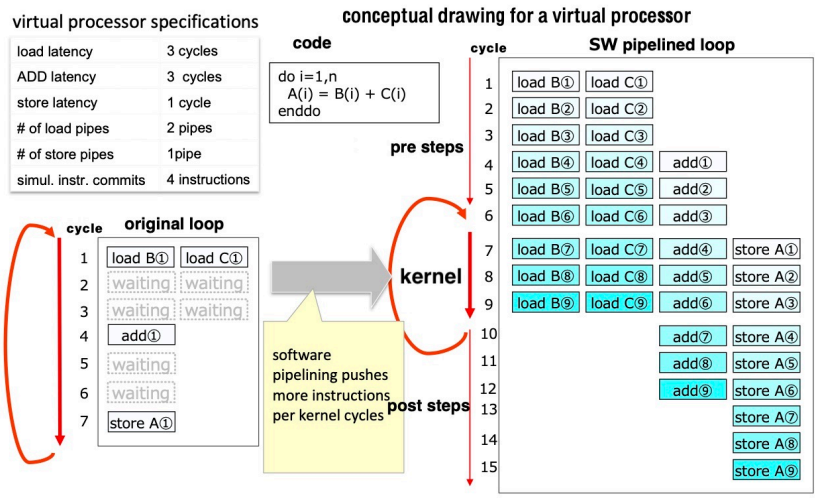


Figure 4.1: Software pipelining (SWPL).

The performance-optimization technique outlined above is known as *instruction scheduling via software pipelining*. Both the K computer and the FX100 were equipped with 128 registers. Compilers could take advantage of this plentiful register pool to cram the kernel sections of programs chock full of load, arithmetic, and store instructions, applying SWPL techniques to generate efficient object codes and achieve high performance. Indeed, one might say that the objective of these machines was to pursue improved performance via in-order strategies.

In contrast, Fugaku has only 32 registers, significantly reducing the number of instructions that can be squeezed into program sections. Attempts to *overfill* programs with more instructions than can be reasonably accommodated tend to produce *register spills*—data leakage from registers to cache—that dramatically degrade performance. This complicates the task of scheduling instructions via SWPL, thus threatening reduced performance in the absence of alternative tuning techniques.

Fugaku's strategy for addressing this situation is to offer an enhanced arsenal of *out-of-order* (OoO) resources to compensate for the loss of registers. Out-of-order execution is a paradigm in which judicious decisions regarding the initiation, execution, and termination of instructions are made to enable automated parallel processing *in hardware* using multiple computational resources and based on factors such as data dependency relationships. Fugaku's basic strategies for improving CPU performance are to develop ARM-ready compilers (with SIMD extensions) while continuing to implement scheduling via software pipelining, to take advantage of the extended out-of-order execution capabilities of the hardware, to improve SWPL scheduling, to develop other simple scheduling techniques, and to use all of these approaches for application performance tuning.

4.2 Comparison of out-of-order resources between Fugaku and Intel machines

Although Fugaku boasts an increase in OoO resources, Intel machines offer a greater wealth of resources overall. We used the RIKEN simulator to conduct performance comparisons with OoO resources and latencies set comparable to those of Intel machines. The application used for this assessment is a computation-heavy code that runs in cache and has been fully performance-tuned for execution on Fugaku. The results are shown in Figure 4.2.

We compared four cases with varying instruction latencies and OoO resources: Fugaku as is, Fugaku enhanced to Haswell levels, Fugaku enhanced to Skylake levels, and Fugaku enhanced beyond Skylake levels. Our findings indicate that enhancing to Haswell levels improves performance by around 30%. This conclusion does not assume the OoO resources are fully on par with those of Intel machines and is the result of a RIKEN simulator evaluation, so it should be kept in mind as a reference assessment. Nevertheless, trends like these can arise in case studies involving arithmetic-heavy in-cache applications.

type	symbol	inst. latency	ROB	IQ	time [ms] raw	time [ms] 1000 cycles	GFLOPS	GB/s
Fugaku default setting	Fugaku	9	128	64	0.00099	0.99	183.96	1.5
10.haswell_lat5_ROB192_IQ60	Haswell	5	192	60	0.000762	0.762	239.01	1.95
30.skylake_lat5_ROB224_IQ97	Skylake	5	224	97	0.000762	0.762	237.76	1.95
31.skylake_lat5_ROB256_IQ128	Skylake (	5	256	128	0.000765	0.765	238.07	1.94

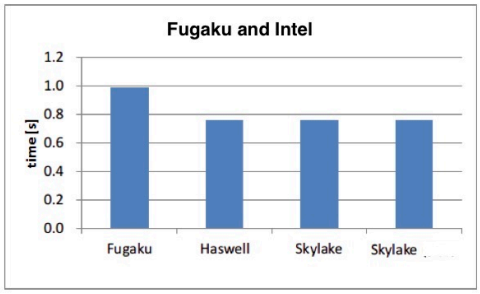


Figure 4.2: Comparing Fugaku's out-of-order resources to those of Intel machines.

Chapter 5

Techniques for tuning single-CPU performance on Fugaku

In this section, we discuss techniques for tuning the single-CPU performance of Fugaku applications by presenting several methods known at the present time. Since the content of this discussion assumes the use of present-day compilers, its relevance and accuracy can be affected by future compilers.

5.1 How to read CPU performance-analysis reports

We begin by discussing how to interpret CPU performance-analysis reports to gain insight into program behavior. Each CPU performance-analysis report presents a bar chart like that shown in Figure 5.1. In this graph, the narrow solid-colored bars in the leftmost column indicate busy times for various hardware components, as discussed in Section 1.3, while the wider bars in the remaining columns are thread-specific *cycle-accounting* displays, in which color-coded segments indicate the contribution of various instruction-commit times and wait times to the thread's overall execution time. Light purple segments indicate functional unit wait times, with large values signifying poor instruction scheduling. Light pink, dark pink, and red segments respectively indicate wait times for accessing the L1D cache, L2 cache, and main memory. These offer at-a-glance insight into the cache and main-memory latency. Magenta segments indicate barrier-synchronization wait times, which are significant when threads are imbalanced.

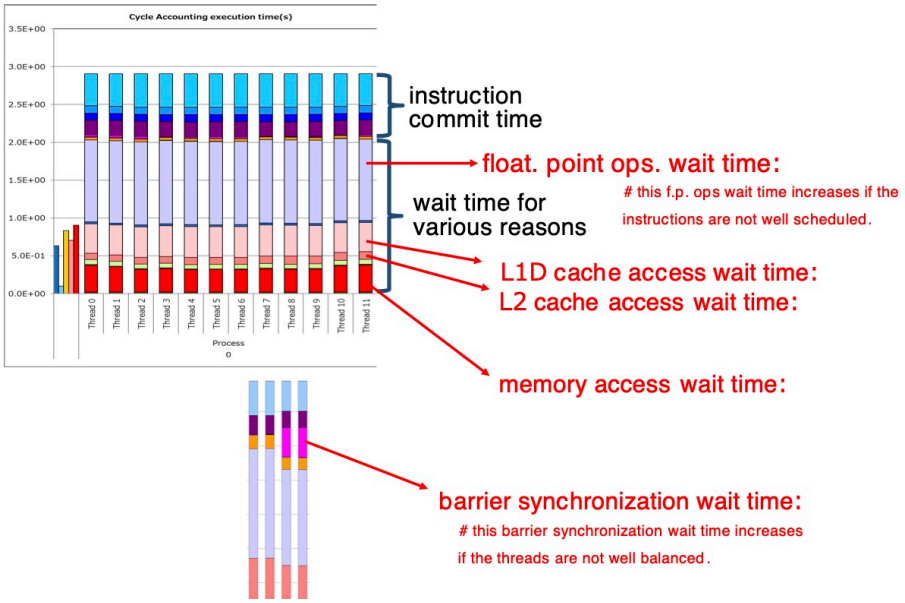


Figure 5.1: Interpretation of CPU performance-analysis reports.

5.2 Tuning technique 1: Optimizing instruction scheduling

Our first tuning technique is a general-purpose method relevant for applications in multiple subcategories, including applications with both low and high B/F ratios. The specific subcategories for which this method can be expected to be effective are shown in Figure 5.2. The schematic diagram shown in Figure 5.3, which indicates various steps in the method as applied to a typical program, also shows that this is a general-purpose method applicable to a wide range of applications. These steps are briefly summarized below and discussed in more detail in the following sections.

1. **Loop fusion** Starting with a stencil computation involving a three-fold nested loop, we can fuse the two inner loops to yield a two-fold nested loop (Figure 5.3, upper left) or fuse all three loops to yield a single loop (Figure 5.3, lower left).

2. **Blocking** In anticipation of the loop-subdivision step below, we apply blocking to the new, elongated loops produced by loop fusion, taking care to ensure that the innermost loop remains sufficiently long. This allows localization of the working arrays is needed to pass information between loops after the loop-subdivision step.

3. **Loop fusion and multithreading** If the outermost loop is too short to allow multithreading, it is fused with the next loop created by blocking to achieve a loop length sufficient for multithreading.

4. **Automatic and manual loop splitting** For loops with long loop bodies, there can be too few registers to allow instruction scheduling. In this case, we must split loops to achieve shorter loop bodies, thereby reducing the number of registers needed. Loop splitting can be performed automatically or manually.

5. **Prefetching** If hardware prefetching is not enabled, we enable software prefetching or introduce explicit software-prefetch instructions into the source code.

6. **Promoting SWPL** SWPL is an important approach to instruction scheduling. A number of techniques are used to make maximal use of this capability.

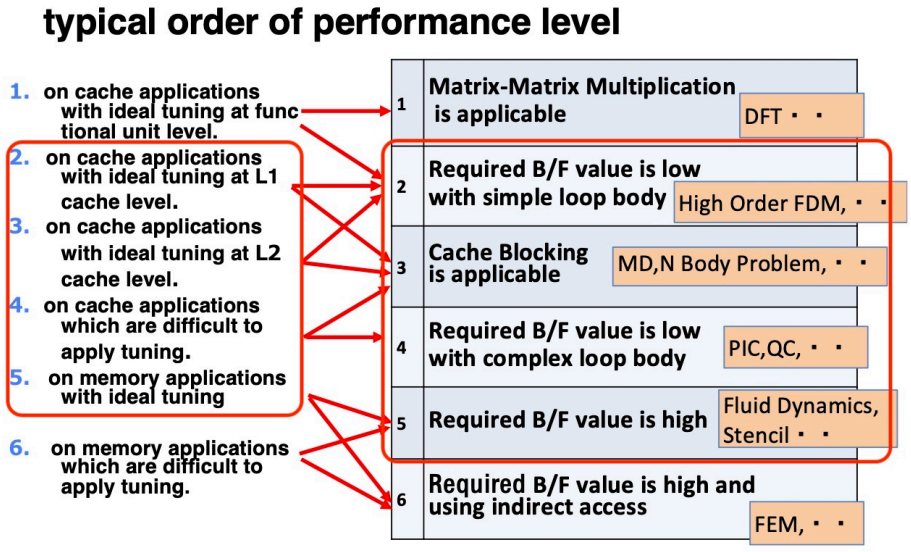


Figure 5.2: Application subcategories to which tuning technique 1 is applicable.

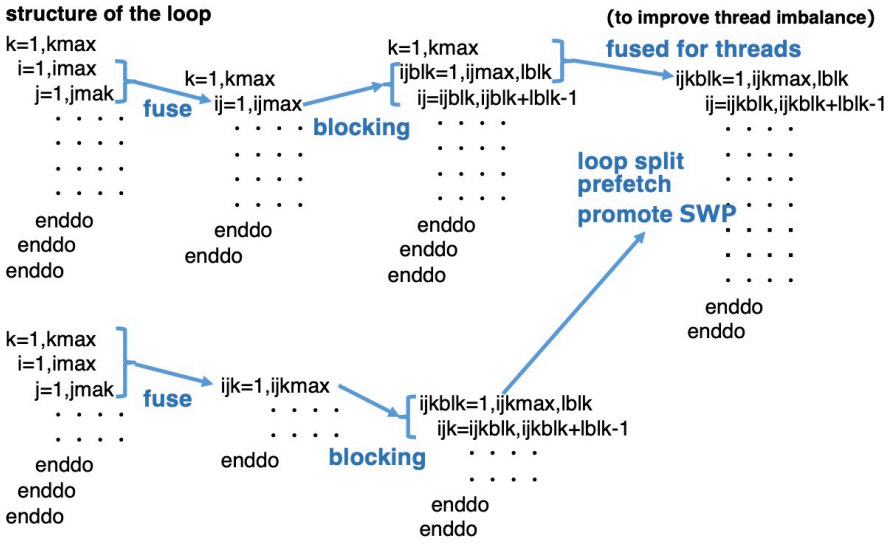


Figure 5.3: Outline of various steps in Tuning Technique 1.

5.2.1 Promoting SWPL and increasing cache efficiency: Loop splitting/loop fusion and blocking

In this section, we discuss the use of loop splitting/loop fusion and blocking as tuning methods to promote SWPL and increase cache efficiency. As mentioned above, Fugaku has 32 registers, which is considerably fewer than the 128 registers of the K-computer and the FX100. For this reason, when codes written for the K-computer or FX100 are executed without modification on Fugaku, there often arises a shortage of registers for SWPL and other scheduling purposes, thus causing register spills and other performance-degrading problems. An effective method for addressing this issue is to split loops to reduce the number of registers used within loop bodies.

However, even loops that have been split cannot be successfully scheduled if the loop length is insufficient. In such cases, fusing loops to increase loop length is an effective tuning strategy. On the other hand, the use of loop fusion and loop splitting requires data transfers between subdivided loops, which—for loops that are *too* long—tend to degrade performance due to the large number of memory accesses involved. This prompts the use of loop blocking as an effective tuning method for ensuring the innermost loops are long enough to allow adequate scheduling after loop fusion.

To demonstrate these ideas, we apply them to the task of tuning a three-fold nested loop. The original (untuned) and tuned programs are listed in Figures 5.4 and 5.5. The performance effect of this tuning is displayed graphically in Figure 5.6 and tabulated quantitatively in Figure 5.7.

① `r4_tune02`

The array-dimension rearrangement, loop splitting, prefetching, and other tuning steps accelerate the tuned code to levels higher than the original version. Because the number of iterations of the innermost loop is rather small (128), SWPL is not applied, and there is some functional unit wait time. The outermost loop over `k` is parallelized, but the region of `private` arrays and the region used for one-dimensional (1D) loop splitting are both relatively small.

② `r4_collapsed.tune02`

The two inner loops over `i` and `j` are fused into a single loop over `ij` to promote SWPL. As this yields a large iteration count for the `ij` loop (16,900), we parallelize this loop over 12 threads, each executing 16900/12=1408 iterations. The application of SWPL reduces functional unit wait time relative to ①. However, because accesses terminate every 1408 iterations, the efficiency of prefetching is degraded, thus increasing the cache wait time.

③ `r4_collapsed.tune03`

Multithreading was restored to the loop over `k` to improve prefetching efficiency, but the large iteration count of the `ij` loop increased the sizes of the working arrays used to pass data between loops, thus increasing main-memory accesses and degrading performance. Therefore, we perform blocking (with block size 128) to localize local arrays and reduce main-memory accesses. We also improve the compiler's optimization capabilities to allow the application of SWPL to loops with as few as 128 iterations. Furthermore, only identical calculations are aggregated and subdivided to reduce working arrays. As a side effect of the modified sequence of arithmetic operations, the number of FMA instructions increases while the number of ordinary floating-point instructions and the overall instruction count decrease. Compared to the untuned program, the performance of this fully tuned code more closely approaches the theoretical performance limit set by the L2-cache busy time.

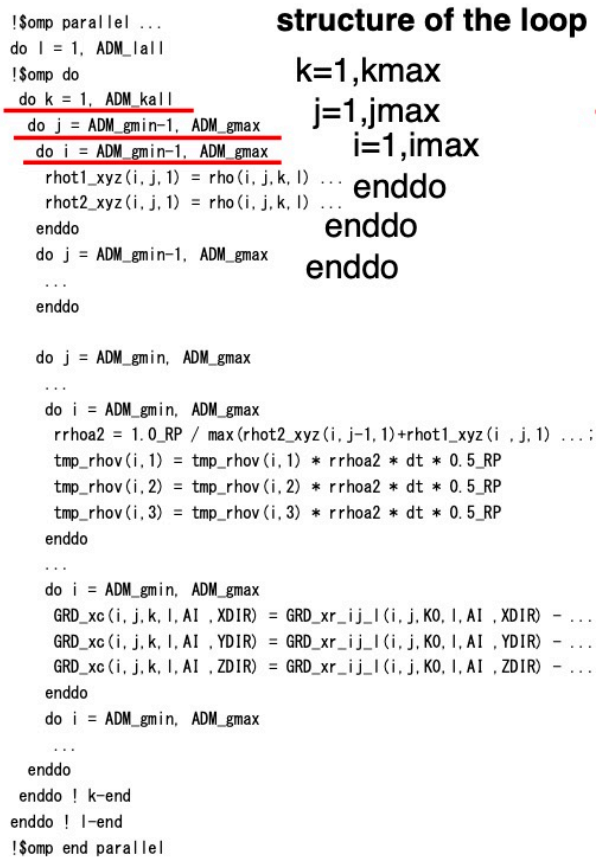


Figure 5.4: Asis (untuned) version of loop splitting/loop fusion and blocking.

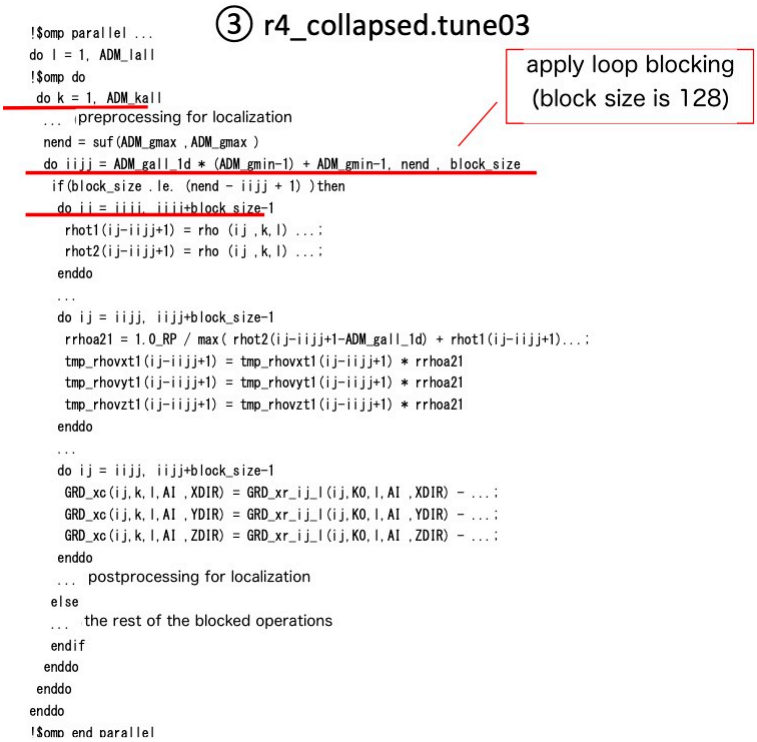


Figure 5.5: Tuned version of loop splitting/loop fusion and blocking (`r4_collapsed.tune03`).

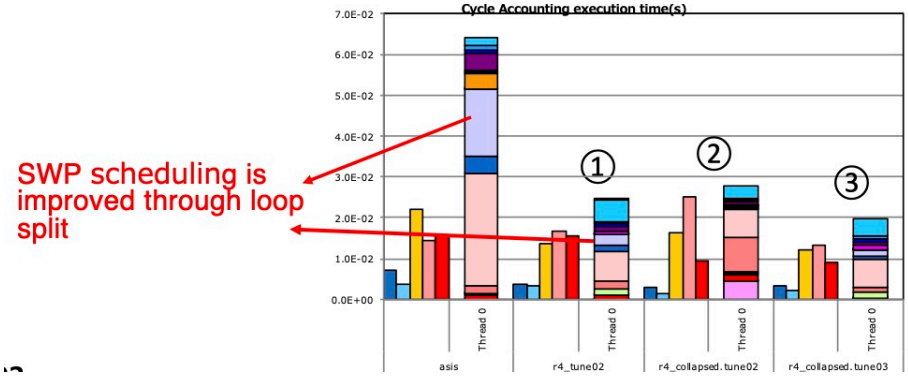


Figure 5.6: Performance charts showing the effect of loop splitting/loop fusion and blocking.

5.2.2 Automatic loop splitting: A first example

Fujitsu developed automated loop-splitting functionality for the Fugaku project. This mechanism supersedes the previous approach—in which loops were split at user-specified branch points indicated by control lines in the source code—with a new technique for automatically identifying appropriate branch points. In this section, we demonstrate the use of automated loop-splitting to tune a meteorology code for modeling microphysics processes. The original (untuned) and final (fully tuned) codes are listed, in abbreviated form, in Figures 5.8 and 5.9. The performance levels of the original, partially tuned, and fully tuned codes are displayed graphically in Figure 5.10 and tabulated quantitatively in Figure 5.11. From the code listing of Figure 5.8, we can see that the loop body is extremely long, thus prohibiting the effective application of SWPL. Additionally, because the code uses both single- and double-precision floating-point values, only 8-SIMD instructions are applied, thus resulting in large numbers of register spill/fills.

source code version	asis	tune02	r4_collapsed. tune02	r4_collapsed. tune03
tuning contents	–	loop split + dimension move	loop fusion	loop fusion + loop blocking
compier version	tcsu1.2.27b	tcsu1.2.27b	tcsu1.2.27b	tcsu1.2.27b
CPU frequency(GHz)	2.0	2.0	2.0	2.0
kernel coded	yes	yes	yes	yes
float. point precision	single	single	single	single
gathering thread #	0	0	0	0
elapsed time[s]	6.40E−02	2.45E−02	2.77E−02	1.97E−02
GFLOPS/process	32.26	85.62	79.31	110.94
memory throughput (R+W) GB/s/proce	ss 62.03	163.78	86.91	119.57
float. point ops/thread	1.72E+08	1.75E+08	1.83E+08	1.82E+08
executed instructions/thread	3.26E+07	5.19E+07	3.02E+07	4.22E+07
load/store instructions/thread	1.08E+07	2.20E+07	1.33E+07	2.03E+07
L1D misses/thread	1.46E+06	1.69E+06	2.96E+06	1.66E+06
L2 misses/thread	8.77E+05	8.76E+05	5.17E+05	4.15E+05
L1 busy ratio	34.14%	55.55%	58.64%	63.18%
L2 busy ratio	22.55%	68.54%	91.08%	67.44%
memory busy ratio	24.23%	63.98%	33.95%	46.71%

Figure 5.7: Performance data of loop splitting/loop fusion and blocking.

In the tuned code of Figure 5.9, automatic loop splitting has been applied at label `loop_fission_target`. We specify the option `-Kloop_fission_threshold=40` to set the loop-splitting threshold to 40. This option can be left unspecified, in which case the loop-splitting precision is determined automatically. Here, we manually set the precision to 40, which is a value that has produced good performance for multiple floating-point precisions. Applying only this loop-splitting optimization yields a partially-tuned version of the code named `tune1`. To complete the tuning, we further apply a blocking step to address the growth in the working arrays expected to accompany loop splitting. This yields our final fully-tuned code, `tune2`. From Figure 5.10, we see that the partially-tuned `tune1` code achieves a reduction in functional unit wait time, but the benefit is partly offset by increased wait times for main-memory and cache accesses. Overall, `tune1` reduces total execution time by 35% compared to the unmodified code. The fully-tuned `tune2` code improves both memory and cache wait times and achieves a 62% reduction in execution time versus the unmodified code.

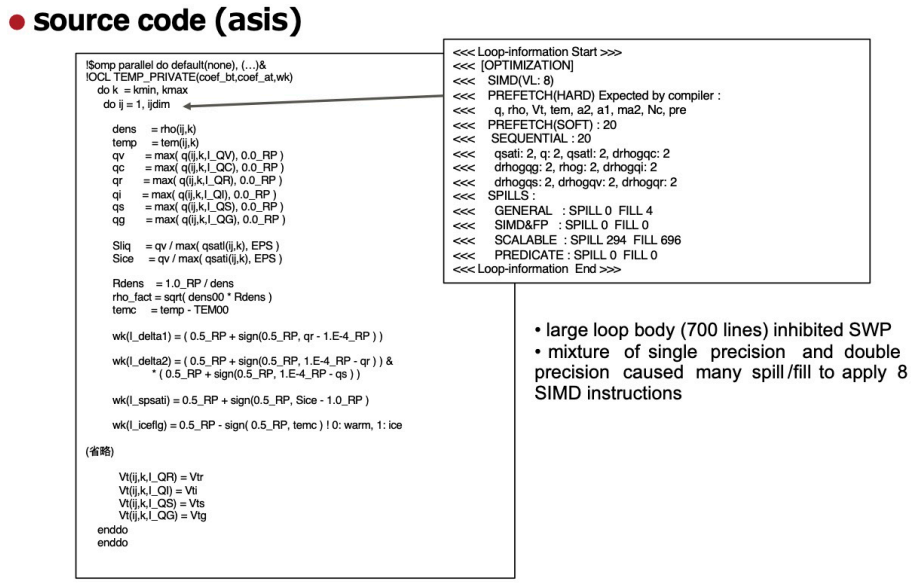


Figure 5.8: Asis version of test code showing automatic loop splitting.

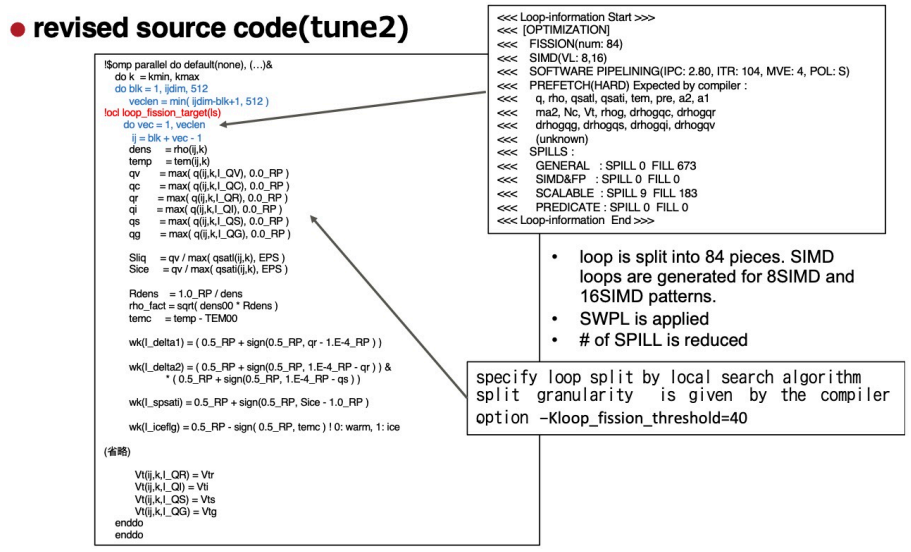


Figure 5.9: tuned version of the test code (`tune2`)

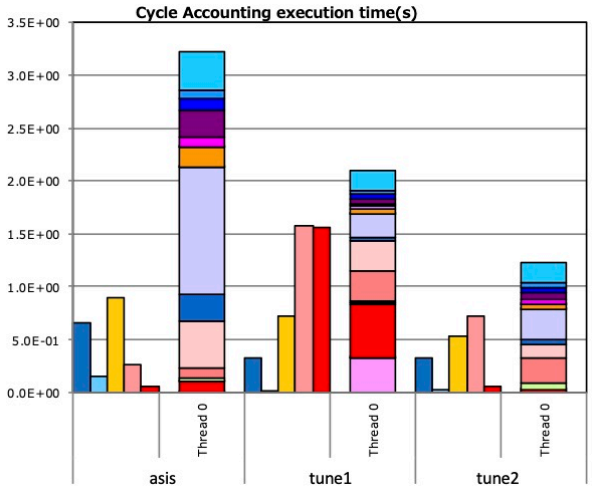


Figure 5.10: Performance comparison of three versions

source code version	asis	tune1	tune2
tuning contents	–	auto loop blocking	auto loop split + blocking
compier version	tcsu1.2.26	tcsu1.2.26	tcsu1.2.26
frequency(GHz)	2.0	2.0	2.0
kernel coded	yes	yes	yes
float. point precision	single	single	single
gathering thread #	0	0	0
elapsed time[s]	3.23E+00	2.09E+00	1.23E+00
GFLOPS/process	101.63	86.91	148.41
memory throughput(R+W) GB/s/proce	4.74	190.58	13.30
float. point ops/thread	2.79E+10	1.55E+10	1.55E+10
executed instructions/thread	4.40E+09	1.98E+09	2.12E+09
load/store instructions/thread	1.57E+09	7.55E+08	8.41E+08
L1D misses/thread	3.02E+07	1.32E+08	9.44E+07
L2 misses/thread	3.67E+06	9.07E+07	3.90E+06
L1 busy ratio	28.23%	34.55%	44.87%
L2 busy ratio	8.04%	75.33%	58.61%
memory busy ratio	1.85%	74.44%	5.19%

Figure 5.11: Performance data for the three code versions.

5.2.3 Automatic loop splitting: A second example

As a second example of automatic loop splitting, we next consider a fluid-modeling code based on hierarchical Cartesian meshes, focusing on the portion of this code responsible for advection computations. Figure 5.12 shows the performance of the code before tuning (far left) and after tuning via automatic loop splitting at 10 distinct values of the splitting threshold (`-Kloop_fission_threshold`) ranging from 10 to 100 (indicated by x-axis labels). For threshold values between 40 and 50, the automated loop-splitting algorithm yields performance on par with that achieved by manually splitting into 31 subdivided loops. In passing, we note that this data set was assembled based on program run times obtained with slightly older compilers and RIKEN simulators, so it may be somewhat out of date.

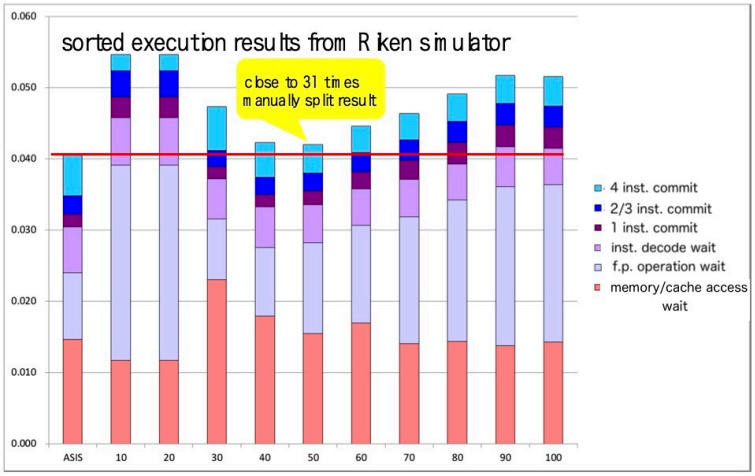
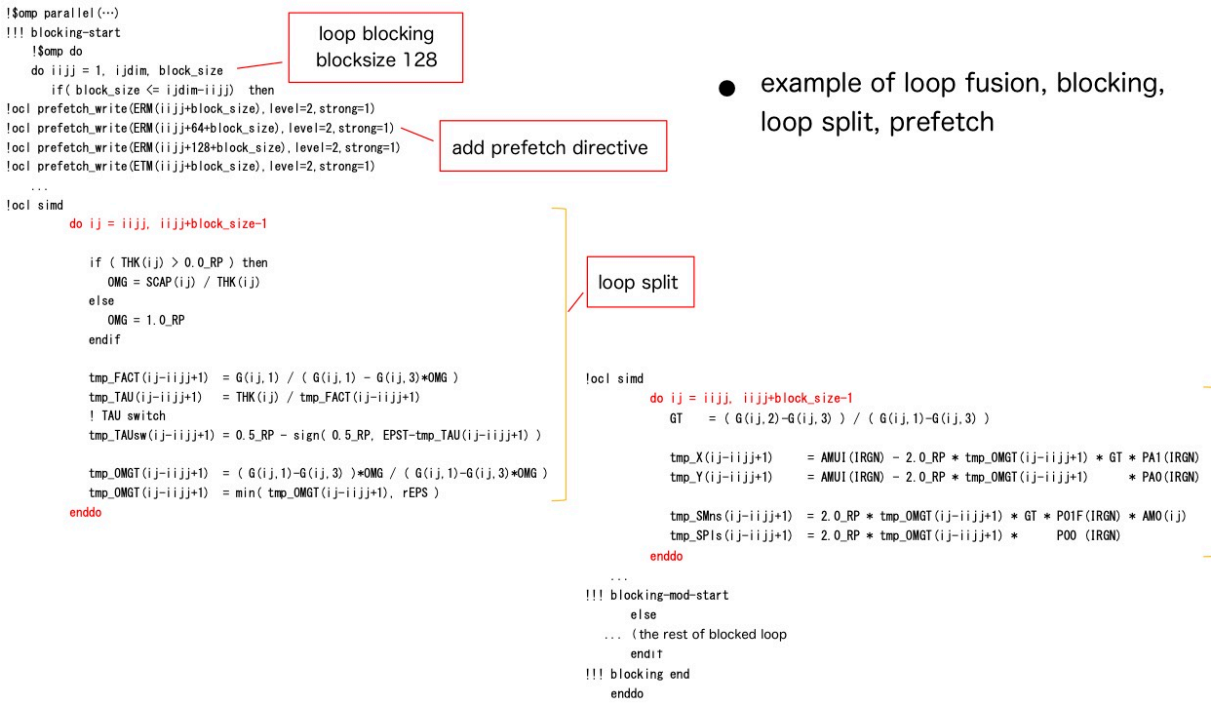


Figure 5.12: Performance charts of automatic loop splitting at various splitting threshold values

5.2.4 Promoting SWPL and enhancing cache efficiency via loop fusion and blocking, loop splitting, and prefetching

Our next example demonstrates how a stencil-based computation can be tuned via loop fusion and blocking, loop splitting, and prefetching to promote SWPL and increase cache efficiency. The final code produced by the tuning process in this case is outlined in Figure 5.13 (we omit a listing of the original code), while the performance of the code before and after tuning is displayed graphically in Figure 5.14 and tabulated quantitatively in Table 5.15. The tuning process uses loop splitting to achieve SWPL, as well as improving wait times for main-memory and cache access. For this reason, we perform loop blocking with a block size of 128 and add prefetch directives to the loop code. From Figure 5.14, we see that the application of SWPL improves scheduling and significantly reduces functional unit wait time, thereby yielding a 40% reduction in overall execution time. Also noteworthy is the increase in the L1-cache reuse and busy time compared to the unmodified program, both reducing L2-cache busy time—which sets the theoretical lower bound on overall runtime in this case—and pushing down the actual runtime to approach this limit more closely than was possible before tuning.




```
!$omp parallel(…)
!!! blocking-start
    !$omp do
    do iijj = 1, ijdim, block_size
       if( block_size <= ijdim-iijj)  then
!ocl prefetch_write(ERM(iijj+block_size),level=2,strong=1)
!ocl prefetch_write(ERM(iijj+64+block_size),level=2,strong=1)
!ocl prefetch_write(ERM(iijj+128+block_size),level=2,strong=1)
!ocl prefetch_write(ETM(iijj+block_size),level=2,strong=1)
    ...
!ocl simd
        do ij = iijj, iijj+block_size-1

           if ( THK(ij) > 0.0_RP ) then
              OMG = SCAP(ij) / THK(ij)
           else
              OMG = 1.0_RP
           endif

           tmp_FACT(ij-iijj+1)  = G(ij,1) / ( G(ij,1) - G(ij,3)*OMG )
           tmp_TAU(ij-iijj+1)   = THK(ij) / tmp_FACT(ij-iijj+1)
           ! TAU switch
           tmp_TAUsw(ij-iijj+1) = 0.5_RP - sign( 0.5_RP, EPST-tmp_TAU(ij-iijj+1) )

           tmp_OMGT(ij-iijj+1)  = ( G(ij,1)-G(ij,3) )*OMG / ( G(ij,1)-G(ij,3)*OMG )
           tmp_OMGT(ij-iijj+1)  = min( tmp_OMGT(ij-iijj+1), rEPS )
        enddo
```

```
!ocl simd
        do ij = iijj, iijj+block_size-1
           GT    = ( G(ij,2)-G(ij,3) ) / ( G(ij,1)-G(ij,3) )

           tmp_X(ij-iijj+1)     = AMUI(IRGN) - 2.0_RP * tmp_OMGT(ij-iijj+1) * GT * PA1(IRGN)
           tmp_Y(ij-iijj+1)     = AMUI(IRGN) - 2.0_RP * tmp_OMGT(ij-iijj+1)      * PA0(IRGN)

           tmp_SMns(ij-iijj+1)  = 2.0_RP * tmp_OMGT(ij-iijj+1) * GT * PO1F(IRGN) * AMO(ij)
           tmp_SPls(ij-iijj+1)  = 2.0_RP * tmp_OMGT(ij-iijj+1) *      PO0 (IRGN)
        enddo
    ...
!!! blocking-mod-start
        else
    ... (the rest of blocked loop
        endif
!!! blocking end
    enddo
```

Figure 5.13: example of loop fusion, blocking, loop splitting, and prefetch (tune).

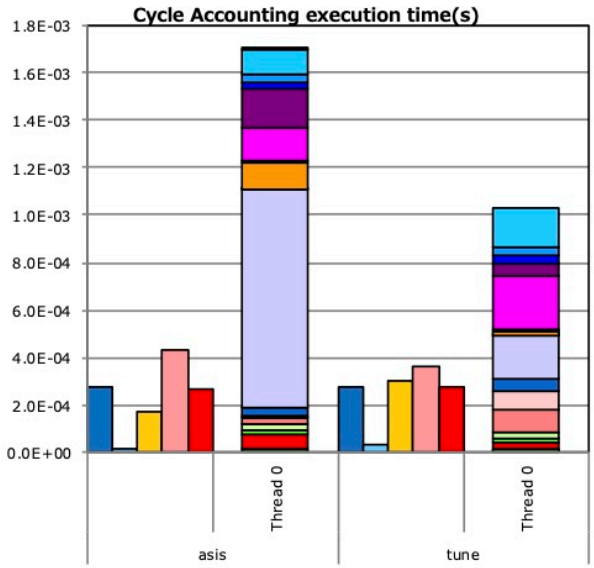


Figure 5.14: Performance chart for the example code before and after tuning.

5.2.5 Adding prefetch directives

When array elements are accessed sequentially, hardware prefetching is typically in effect. However, even in the computational stencils whose arrays are sequentially accessed arrays, there can be cases which exhibit long wait times for cache access, thereby resulting in high rates of cache misses and demand misses for the L1D and L2D caches. The miss rate of a cache is an aggregate of three contributions: the miss rate due to hardware prefetching, the miss rate due to software prefetching, and the demand miss rate. Together these three factors

source code version	asis	tune
tuning contents	–	loop fusion blocking loop split prefetch
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	1.70E−03	1.03E−03
GFLOPS/process	136.46	228.98
memory throughput(R+W) GB/s/proce	40.71	68.17
float. point ops/thread	1.93E+07	2.01E+07
executed instructions/thread	1.45E+06	1.97E+06
load/store instructions/thread	2.16E+05	5.73E+05
L1D misses/thread	3.94E+04	4.07E+04
L2 misses/thread	8.34E+03	9.10E+03
L1 busy ratio	10.49%	30.97%
L2 busy ratio	25.32%	35.01%
memory busy ratio	15.90%	26.63%

Figure 5.15: Performance data before and after tuning.

yield 100% of the cache miss rate. Large contributions from the first two factors are harmless, but high values of the demand miss rate can cause problems. An effective method for increasing performance in such cases is to add prefetch directives to the source code to induce the generation of software prefetches.

Figure 5.16 shows an example of a loop tuned by the addition of prefetch directives. In this case, the loop over `ij` is ascending but the loop over `k` is descending, and successive iterations of the `k` loop do not yield sequential array accesses. This destroys the sequential pattern of array accesses and precludes the use of hardware prefetching. We rectify this difficulty by specifying L2-cache prefetch directives to ensure that the array elements to be accessed on the next iteration of the `k` loop are preloaded into the L2 cache.

In the code listing of Figure 5.17, we have added several more prefetch directives to complete the tuning of this loop. We suppose that the loop body involves eight sets of array accesses; then prefetching 256 bytes worth of cache lines amounts to prefetching eight sets $\times$ 256 bytes = 2048 bytes = 2 kB, a data volume that is easily accommodated by Fugaku's 64-kB L1D cache. Since 256 bytes corresponds to 64 single-precision array elements, the directives in Figure 5.17 serve to prefetch the next 64 array elements into the L1D cache.

The performance effect of adding prefetch directives is displayed graphically in Figure 5.18 and tabulated quantitatively in Figure 5.19.

For the untuned code, we find high demand-miss rates of 63.40% and 49.98%, respectively, for the L1D and L2 caches, which indicate problematically large wait times for accessing main memory and cache. The addition of prefetch directives to reduce these demand-miss rates significantly reduces wait times for both main memory and cache—and also reduces L2-cache busy time, thereby yielding an overall reduction of 30% in program execution time. Indeed, the high performance of the tuned code drives down overall runtime to levels approaching the theoretical lower bound given by the L2-cache busy time. The number of executed instructions appears to have increased, but, in fact, the only new instructions generated are prefetch instructions and address-evaluation instructions resulting from the addition of prefetch directives.

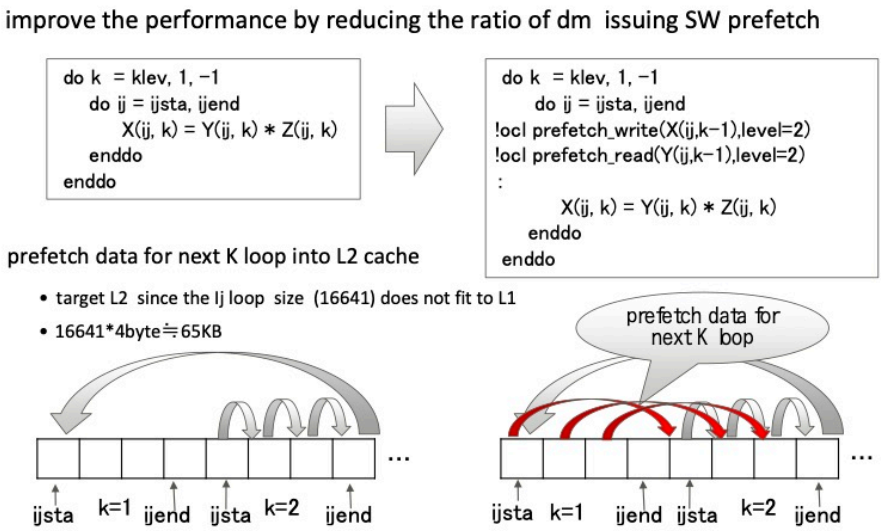


Figure 5.16: A two-fold nested loop partially tuned by adding two prefetch directives, with diagrams illustrating the effect of the directives on the L2 cache.

```
do k  = klev, 1, −1
   do ij = ijsta, ijend
!ocl prefetch_write(X(ij+64,k),level=1)
:
        X(ij, k) = Y(ij, k) * Z(ij, k)
   enddo
enddo
```

```
3955       do ij = ijsta, ijend
3956 !ocl prefetch_write(RL(ij,k-1),level=2)
3957 !ocl prefetch_write(SL(ij,k-1),level=2)
3958 !ocl prefetch_read(SL(ij,k),level=2)
3959 !ocl prefetch_read(RL(ij,k),level=2)
3960 !ocl prefetch_read(RE(ij,k-1),level=2)
3961 !ocl prefetch_read(TE(ij,k-1),level=2)
3962 !ocl prefetch_read(SER(ij,k-1),level=2)
3963 !ocl prefetch_read(SET(ij,k-1),level=2)
3964 !ocl prefetch_write(RL(ij+64,k),level=1)
3965 !ocl prefetch_write(SL(ij+64,k),level=1)
3966 !ocl prefetch_read(SL(ij+64,k+1),level=1)
3967 !ocl prefetch_read(RL(ij+64,k+1),level=1)
3968 !ocl prefetch_read(RE(ij+64,k),level=1)
3969 !ocl prefetch_read(TE(ij+64,k),level=1)
3970 !ocl prefetch_read(SER(ij+64,k),level=1)
3971 !ocl prefetch_read(SET(ij+64,k),level=1)
3973          recip = 1.0_RP / ( 1.0_RP - RL(ij,k+1)*RE(ij,k) )
3974
3975          RL(ij,k) = RE (ij,k) + TE(ij,k) * ( RL(ij,k+1)*TE (ij,k)            ) * recip
3976          SL(ij,k) = SER(ij,k) + TE(ij,k) * ( RL(ij,k+1)*SET(ij,k) + SL(ij,k+1) ) *
recip
3977       enddo
3979    enddo
```

Figure 5.17: The program of Figure 5.16 after adding additional prefetch directives.

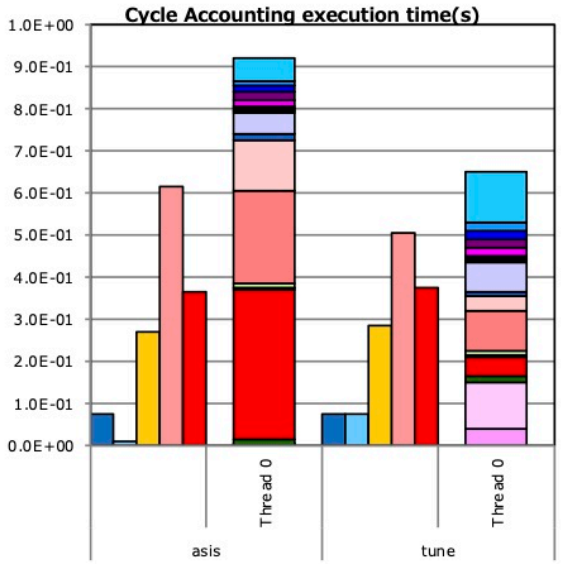


Figure 5.18: Performance charts showing the effect of adding prefetch directives.

source code version	asis	tune
tuning contents	–	add prefetch directive
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	9.20E−01	6.49E−01
GFLOPS/process	74.90	106.13
memory throughput(R+W) GB/s/proce	101.36	147.05
float. point ops/thread	5.76E+09	5.76E+09
executed instructions/thread	5.95E+08	1.18E+09
load/store instructions/thread	1.83E+08	1.89E+08
L1D misses/thread	5.10E+07	4.04E+07
L2 misses/thread	2.29E+07	2.44E+07
L1D miss dm ratio	63.40%	18.18%
L2 miss dm ratio	49.98%	17.30%
L1 busy ratio	30.42%	45.01%
L2 busy ratio	66.71%	78.08%
memory busy ratio	39.59%	57.44%

Figure 5.19: Performance data showing the effect of adding prefetch directives.

5.2.6 Using multiple structure load/store instructions

Nested loops in which the outermost loop runs over indices of an array, such as the loop in the blue rectangle near the bottom of the code listing on the left side in Figure 5.20, typically generate inefficient `gather load` or `scatter store` instructions. However, if the length of the outermost loop (i.e., the length of the corresponding array dimension) is four or less, it can be possible to use efficient `multiple structure load/store` instructions instead. In this section, we discuss how this observation can be used to tune for performance.

The blue-framed loop in Figure 5.20 effects an in-place component-wise multiplication of the `FXYZ` array by a coefficient array. To this end, elements are both loaded from and stored to `FXYZ` on each loop iteration. The idea behind the tuning strategy is to introduce working arrays that allow the loading and storing operations to be carried out on *separate* arrays, thereby enabling all `gather load` and `scatter store` instructions to be replaced by `multiple structure load/store` instructions. Note that the need to perform this tuning step by hand is a limitation of the current compiler version. In the future, we expect that `multiple structure load/store` instructions will be used automatically, thus obviating the need to rewrite loops by hand, as seen in the example above.

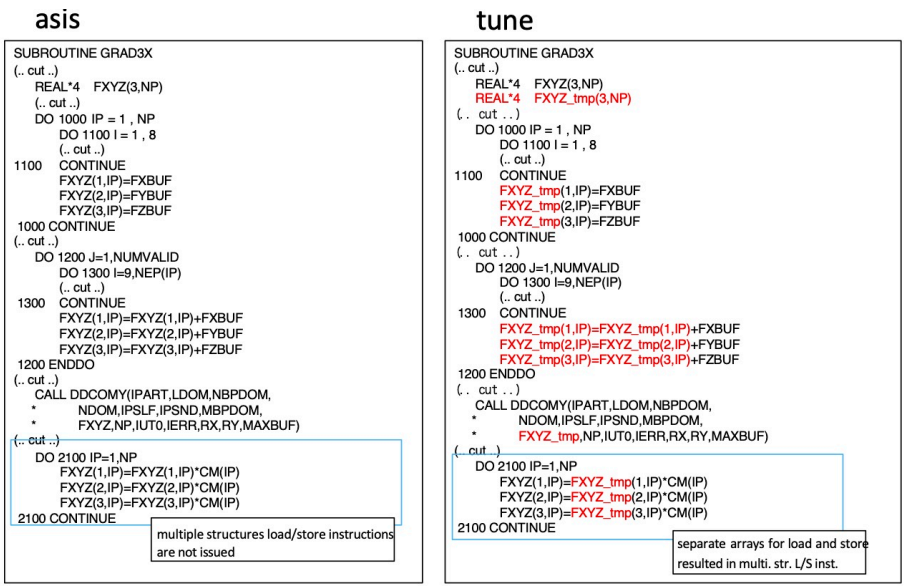


Figure 5.20: Using `multiple structure load` instructions.

The performance of the original and tuned codes is plotted in Figure 5.21 and tabulated in Figure 5.22. For example, whereas the use of `gather load` generates one instruction for each SIMD gather/load of a single element of $\texttt{FXYZ}(1,\cdot,\cdot)$, switching to `multiple structure load` causes just one SIMD instruction to be issued for the full set of elements $\{\texttt{FXYZ}(1,\cdot,\cdot),\cdots,\texttt{FXYZ}(3,\cdot,\cdot)\}$. From Figure 5.22, we see that replacing `gather load/scatter store` with `multiple structure load/store` significantly reduces the number of load/store instructions, while from Figure 5.21, we see that the generation of efficient load/store instructions greatly reduces both the wait time for L1 cache access and cache busy time, thus enabling program performance levels that approach the upper bound set by the main-memory bandwidth. These improvements reduce the overall program runtime from 1.73 to 1.22 seconds.

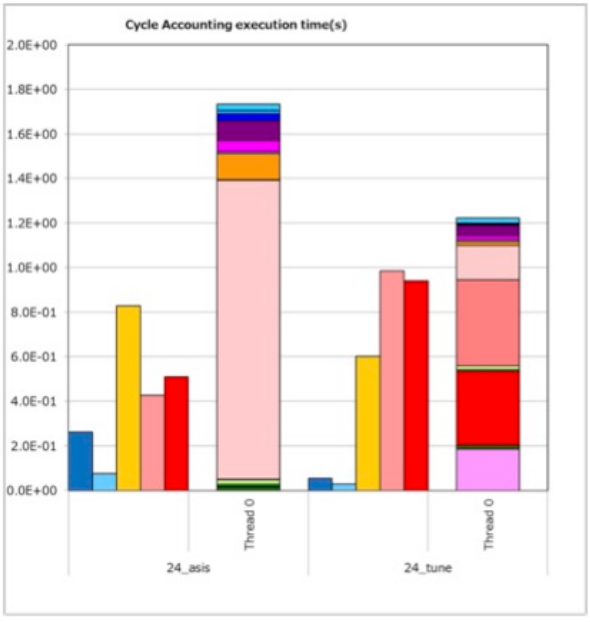


Figure 5.21: performance chart showing the effecgt og `multiple structure load/store (1)`

source code version	asis	tune
tuning contents	–	utilize multiple structures load/store instructions
compier version	tcsds-1.2.27b	tcsds-1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	1.73E+00	1.22E+00
GFLOPS/process	11.45	16.24
memory throughput(R+W) GB/s/process	75.14	196.96
float. point ops/thread	1.66E+09	1.66E+09
executed instructions/thread	6.15E+08	3.32E+08
load/store instructions/thread	2.48E+08	1.10E+08
L1D misses/thread	3.56E+07	6.16E+07
L2 misses/thread	3.62E+07	6.24E+07
L1 busy ratio	47.70%	49.15%
L2 busy ratio	24.57%	80.59%
memory busy ratio	29.35%	76.94%
multiple structures load inst./thread	0.00E+00	3.45E+07
multiple structures store inst./thread	0.00E+00	3.45E+07
gather load inst./thread	1.03E+08	0.00E+00
scatter store inst./thread	1.03E+08	0.00E+00

Figure 5.22: performance data showing the effecgt og `multiple structure load/store (1)` instructions.

A second example is presented in Figure 5.23. Here we have rearranged the dimensions of two arrays: in the upper half of the code, the first array dimension (whose index runs over values 1-6) has become the third dimension, while in the lower half, the first array dimension (with index running from one to three) has become the third dimension. The results of these tuning steps are shown in Figure 5.24, which also shows cycle-accounting charts for the original, untuned code (left), the code after the first array rearrangement (center), and the code after both array rearrangements (right). With previous compilers, rearranging array dimensions often yielded significant performance improvements. However, compilers have become more sophisticated and are now able to auto-generate `multiple structure load` instructions, in some cases involving loops of length four or below. This includes, in particular, the loop in the second half of Figure 5.23, for which the rearrangement we performed by hand above was, in fact, already done automatically by the compiler for the original (untuned) source code. This is why no performance improvement results from going from the center to the right column of Figure 5.24. We conclude that the tuning method of this section is unnecessary for array dimensions of length 1-4, for which `multiple structure load` instructions are generated automatically.

On the other hand, the first of the arrays we rearranged above has a length greater than four, so in this case, the tuning method of this section should improve performance by increasing contiguous memory access. The fact that we see almost no difference between the left and center columns of Figure 5.24 is due to the relatively minimal cost of the upper half of the program in comparison to the lower half. Nevertheless, in general, we recommend tuning via array-dimension reorganization for loops whose first dimension has a length greater than four.

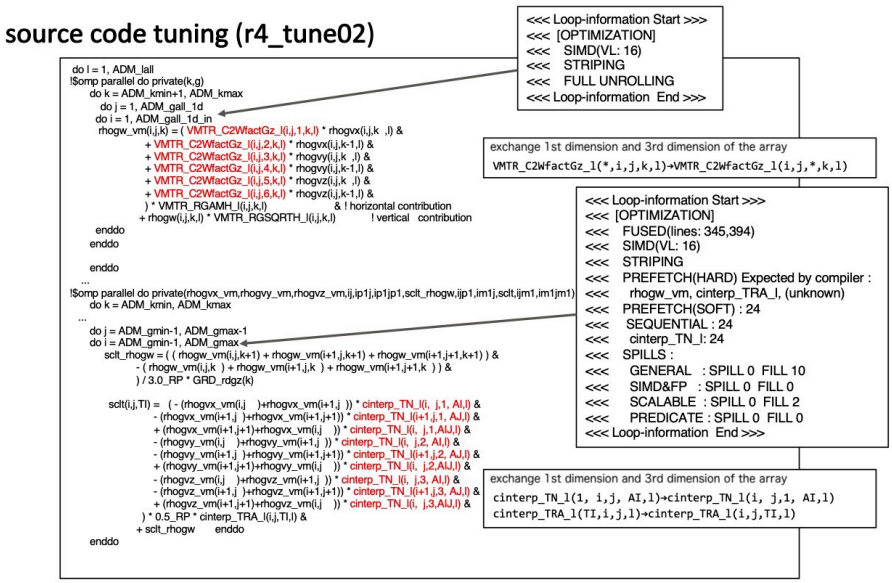


Figure 5.23: example of exploiting `multiple structure load` (2)

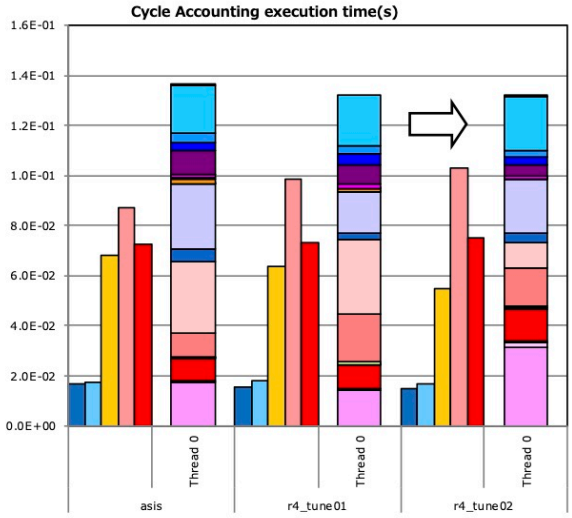


Figure 5.24: Performance chart showing the effect of `multiple structure load` (2)

5.2.7 Using SWP options

In Section 4.1, we discussed SWPL. The cycle distance between the start of instructions for loop iterations i and $i+1$, shown in Figure 4.1, is known as the *initiation interval* (II) (see also Figure 5.25). In those figures, it can be seen that shorter II values allow greater numbers of instructions to be squeezed together at the expense of requiring more registers, while long II values reduce the number of registers required to run a program. However, they also prohibit dense packing of instructions, thereby yielding sparse machine codes and reducing the effect of instruction scheduling.

For the K computer and the FX100, instruction scheduling proceeded by expanding from short-II regions, seeking instruction sequences offering maximal benefit for the available number of registers. If no satisfactory instruction sequences could be found, the scheduling attempt was abandoned. In other words, the SWPL strategy was to seek only a single type of solution and simply to give up if it was not found. For the Fugaku project, the expansion of OoO resources was accompanied by the adoption of a "strong SWPL" philosophy,

in which OoO is used to increase the efficiency of spills due to register shortages. As a result, user-selectable options for applying SWPL—even at the expense of generating register spills—were added to the compiler.

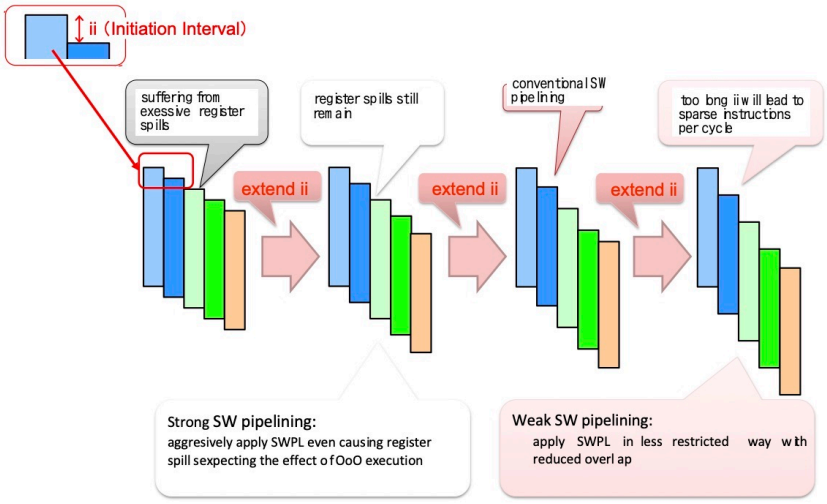


Figure 5.25: Strong SWPL.

An example of strong SWPL is shown in Figure 5.26, which shows an original compiler listing (left) and the same listing after specifying the `swp_strong` option. We see that SWPL has been applied to the listing at the right. The `swp_strong` option is available in FORTRAN and C/C++ TRAD modes. The performance effect of strong SWPL is displayed graphically in Figure 5.55 and tabulated quantitatively in Figure 5.56. In this example, the introduction of SWPL improves functional unit wait time but does not significantly improve performance. Nevertheless, we expect there will be some cases in which the `swp_strong` option has a noticeable effect and have thus included it in this survey of tuning techniques.

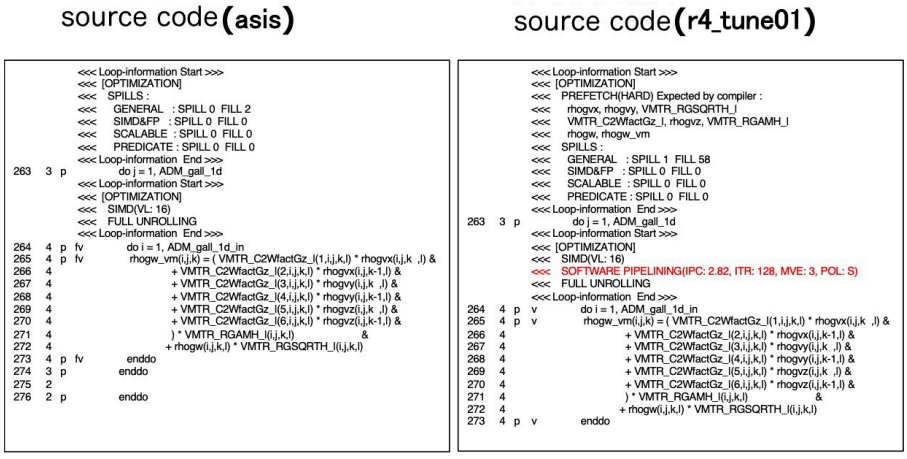


Figure 5.26: An example of strong SWPL.

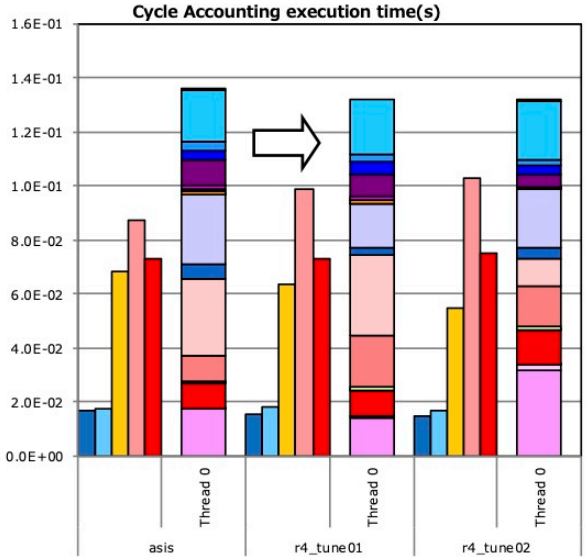


Figure 5.27: Performance charts showing the effect of strong SWPL.

source code version	asis	r4_tune01
tuning contents	–	16 SIMD promotion
compiler version	tcsu1.2.29	tcsu1.2.29
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	1.35E-01	1.32E-01
GFLOPS/process	103.70	106.16
memory throughput(R+W) GB/s/proce	138.49	141.97
float. point ops/thread	1.22E+09	1.22E+09
executed instructions/thread	2.05E+08	2.11E+08
load/store instructions/thread	6.03E+07	6.00E+07
L1D misses/thread	1.02E+07	1.03E+07
L2 misses/thread	4.71E+06	4.77E+06
L1 busy ratio	54.14%	51.02%
L2 busy ratio	64.89%	74.87%
memory busy ratio	54.10%	55.46%
continuous SIMD load inst./thread	2.54E+07	2.54E+07
gather load inst./thread	6.72E+06	6.72E+06
scatter store inst./thread	5.64E+06	5.64E+06

Figure 5.28: Performance data showing the effect of strong SWPL.

5.2.8 Summary of the instruction scheduling tuning

The various tuning techniques discussed in the previous seven subsections are shown in Figure 5.29 and can be summarized as follows:

- Improving functional unit wait times requires instruction-scheduling methods such as SWPL. An effective tool for this purpose is loop splitting to reduce register use. Loop fusion to increase loop lengths can also be effective.
- Cache wait times can be improved by prefetching to reduce cache-access latency.
- An effective technique for improving main-memory-access wait times is blocking loops after loop splitting to reduce the size of working arrays.
- Barrier-synchronization wait times can be reduced by fusing loops to yield loops of longer length and then reducing any thread imbalances that arise.
- Instruction commit times can be reduced by converting non-contiguous data accesses to contiguous accesses, thus increasing the efficiency of load instructions.

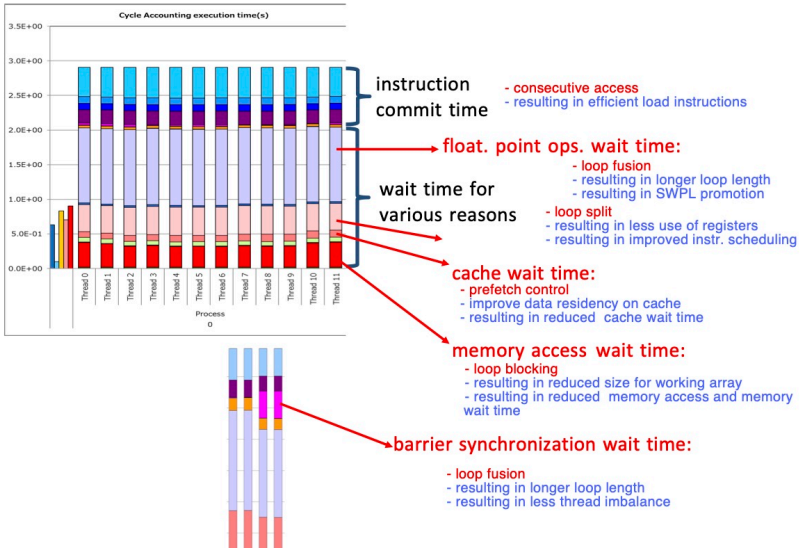


Figure 5.29: Summary of Section 5.2.

5.3 Tuning Technique 2: indirect access applications

The tuning technique discussed in this section is applicable to applications with low B/F ratios that use indirect memory access (Figure 5.30), typically seen in unstructured applications such as FEM. Although applications in this category are considered to be the most difficult of all applications to tune for high performance, for applications to which the tuning technique presented here is applicable, it is possible to achieve ideal performance for in-memory calculations.

5.3.1 Node ordering to improve locality of node indices

Next, we consider the typical sparse matrix-vector multiplication program of Figure 5.31. Here, matrices and vectors are stored in compressed-row storage (CRS) format, so matrix entries are accessed contiguously, while vectors are accessed via lists of indices.

We use the roof-line model to estimate the performance of this program. Since the data used in this estimation are for the K computer, we are really estimating performance for execution on that machine. The roof-line model makes the idealized assumption that list-accessed vector components reside in cache. If we

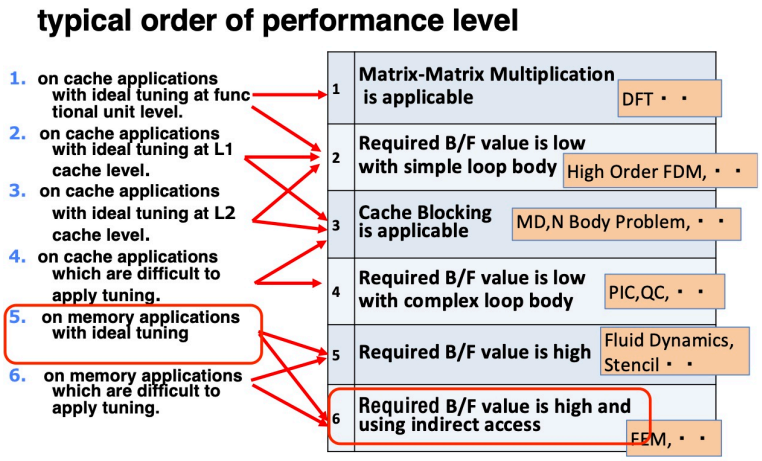


Figure 5.30: Application categories to which Tuning Technique 2 is applicable.

assume that the relevant portions of vectors are present in the L1 cache, we can then entirely neglect main-memory access for vector components, so only matrix entries and list indices require loading from memory. Then, the number of bytes loaded on each inner-loop iteration (for this single-precision program) is 2×4 bytes = 8 bytes. (We neglect the instruction that stores product-vector entries, as this occurs outside the inner loop and is, thus, executed relatively few times.) The inner loop performs one multiplication and one addition, for a total of two floating-point operations. Hence, the B/F ratio for this program is 8/2=4. The theoretical memory bandwidth of the K-computer hardware is 64 GB/s. With an effective memory bandwidth of 46 gigabytes/s and a theoretical CPU performance of 128 gigaflops/s, the effective hardware B/F ratio is 46/128=0.36. Therefore, with the idealized assumption that vector components reside in cache, the peak performance ratio estimated by the roof-line model is the ratio of the effective hardware B/F ratio (0.36) to the program's B/F ratio (4), or 0.36/4=9%. However, actual measurements for the program of Figure 5.31 on the K computer revealed poor performance, with values ranging from 1/5 to 1/10 of this estimate.

```
ICRS=0
    DO 110 IP=1,NP
      BUF=0.0E0
      DO 100 K=1,NPP(IP)
        ICRS=ICRS+1
        IP2=IPCRS(ICRS)
        BUF=BUF+A(ICRS)*S(IP2)
100   CONTINUE
      AS(IP)=AS(IP)+BUF
110 CONTUINE
```

Figure 5.31: Sparse matrix-vector multiplication program.

This poor performance is due to the failure of the assumption that vector components will be present in the L1D cache when accessed. To understand this, it is helpful to visualize the node number relationship between the calculated node and the referenced node. Figures 5.32 shows such relationship plot for tetrahedron element model, and Figure-5.33 for hexahedron element model. For tetrahedron element model, consisted of the approximately 2.7 million nodes in total, each node requires the access to wide range of random access. For hexahedron elements, consisted of the approximately 27.0 million nodes in total, the first 1 million nodes require random access and the other nodes require bipolar pattern access.

To mitigate the phenomenon of index sets distributed effectively randomly over wide ranges, we rearrange the ordering of mesh-node indices to ensure that physically neighboring nodes are assigned to neighboring components of vectors. More specifically, as shown in Figure 5.34, we first segment the physical region into blocks by subdividing each coordinate axis into multiple intervals. We subdivide each block into exterior and interior regions and order nodes within the block from interior to exterior based on physical coordinates.

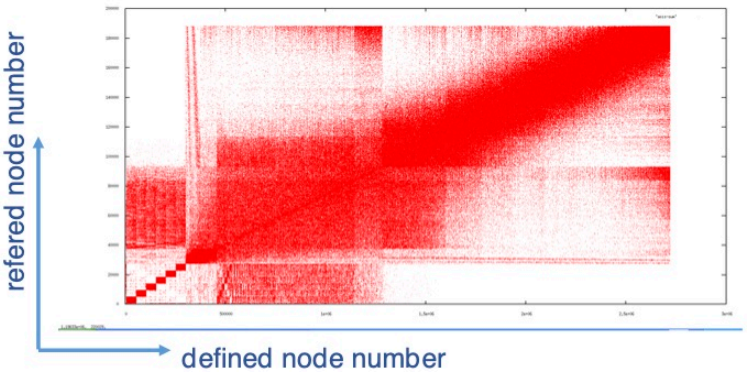


Figure 5.32: node definition and reference plot for the tetrahedron element model

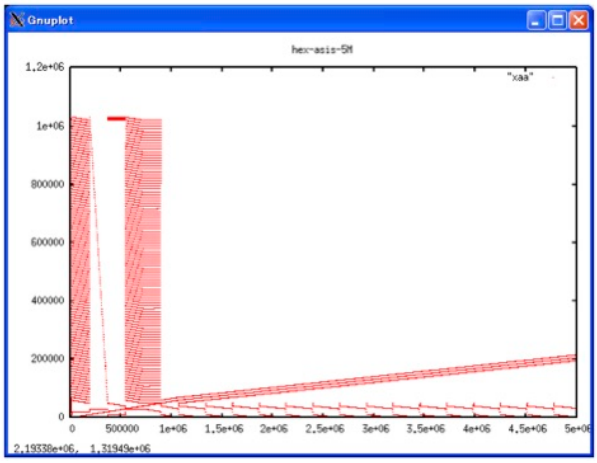


Figure 5.33: node definition and reference plot for the hexahedron element model

Ordering nodes in this way ensures that the indices of the nodes comprising a single element have neighboring values. By adjusting the block sizes, we can arrange for the majority of indirect vector-component accesses to be served by the L1D cache.

After reordering nodes, the component-index plots of Figures 5.32 and 5.33 for the quadrilateral and parallelepiped are transformed into Figures 5.35 and 5.36. In both cases, points appear to be clustered in three lines. However, closer inspection reveals that the random distributions of indices over wide ranges have been successfully localized.

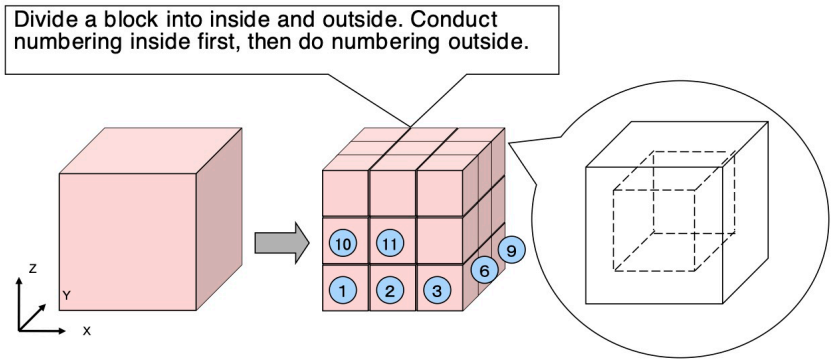


Figure 5.34: Schematic depiction of node-ordering algorithm.

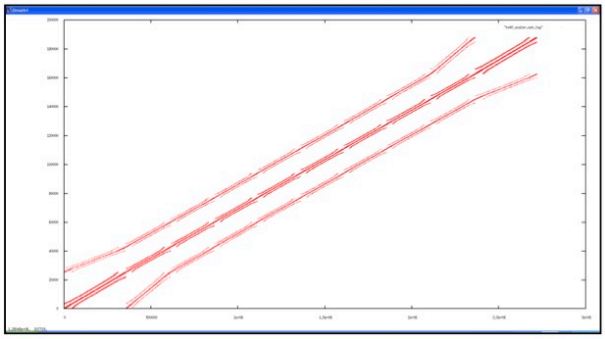


Figure 5.35: node definition and reference relationship plot for tetrahedron element model - after node ordering.

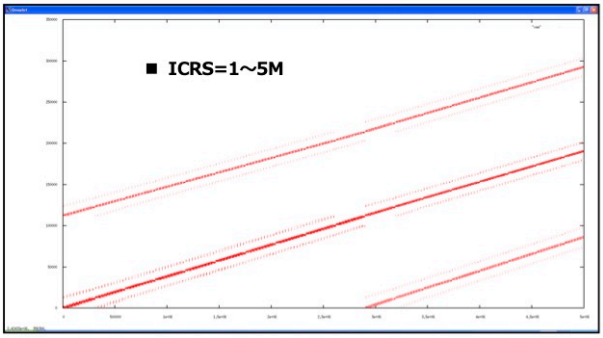


Figure 5.36: node definition and reference relationship plot for hexahedron element model - after node ordering.

Figure 5.37 details performance results obtained on the K computer for the two FEM structures before and after node ordering. For both the tetrahedron model and hexahedron model, the node-ordering optimization increases the performance, achieving the sustained performance of close to 9% which is the theoretical upper bound for K computer estimated on the assumption that all vector components reside in the L1 cache.

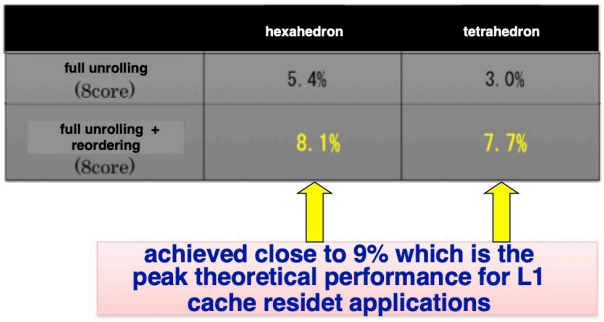


Figure 5.37: K-computer results after node ordering.

Next, the performance effect of node ordering for execution on Fugaku is displayed graphically in Figure 5.38 and tabulated quantitatively in Figure 5.39. The node ordering ensures that many indirect array-element accesses reference data present in cache, thereby reducing L1D cache misses and decreasing wait times for memory and cache access. For hexahedron element model, the number of executed instructions for thread 0 of tuned version increases compared to asis version, but the number of aggregate instructions of all threads reduces. For tetrahedron element model, the number of floating point operations for thread 0 of tuned version increases compared to asis version, but the number of aggregate f.p. operations stay the same. Evidently, performance is significantly improved for both the hexahedron element model and the tetrahedron element model.

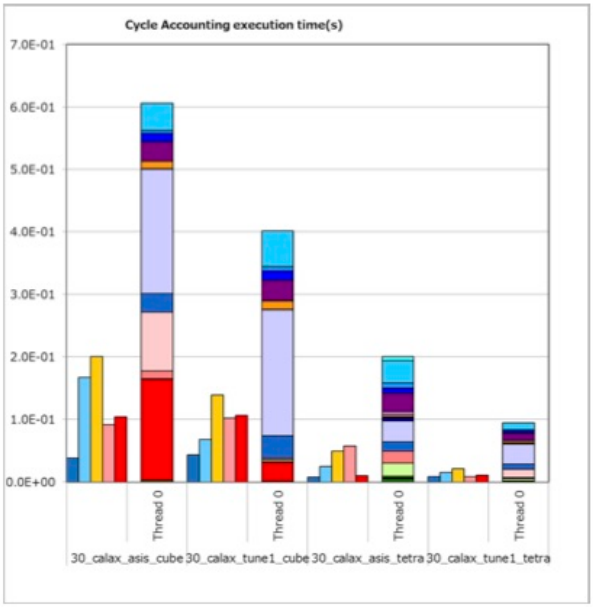


Figure 5.38: Performance charts showing the effect of node ordering for execution on Fugaku.

source code version	asis(hex.)	tune1(hex.)	asis(tetra.)	tune1(tetra.)
tuning contents	asis	ordering	asis	ordering
compiler version	tcsds-1.2.26	tcsds-1.2.26	tcsds-1.2.26	tcsds-1.2.26
frequency(GHz)	2.0	2.0	2.0	2.0
kernel coded	yes	yes	yes	yes
floating point precision	single	single	single	single
gathering thread #	0	0	0	0
elapsed time[s]	6.06E-01	3.91E-01	1.93E-01	9.45E-02
GFLOPS/process	21.93	34.02	10.02	20.43
memory throughput(R+W) GB/s/process	43.86	69.54	13.21	29.32
float. point ops/thread	1.10E+09	1.11E+09	1.52E+08	1.58E+08
executed instructions/thread	5.84E+08	6.21E+08	4.22E+08	1.37E+08
load/store instructions/thread	1.38E+08	1.48E+08	1.25E+08	3.48E+07
L1D misses/thread	1.33E+07	8.97E+06	3.68E+06	9.51E+05
L2 misses/thread	8.10E+06	8.28E+06	6.32E+05	7.77E+05
L1 busy ratio	33.05%	35.60%	25.48%	22.01%
L2 busy ratio	15.11%	26.09%	29.91%	9.42%
memory busy ratio	17.13%	27.16%	5.16%	11.45%

Figure 5.39: performance data showing the effect of node ordering for execution on Fugaku.

5.3.2 apply SIMD via loop re-rolling

The original source code of the loop considered in this section is shown in Figure 5.40. At line 489 and elsewhere, we have manually unrolled a loop over the values 1, 2, and 3 for the first index of the `FXYZ` array. Additionally, at line 474, we have unrolled a loop over values 1-8 for the first index of the `NODE` array, corresponding to nodes in the loop over `NE`. These program sections have not been SIMD applied. We now *re-roll* these manually unrolled loops to restore the loop structures. The resulting code is outlined in Figure 5.41. The length of the innermost loop is set to eight to match Fugaku's SIMD width, but the number of elements actually used is three (for x, y, and z). Iterations 1-8 of the surrounding loop correspond to the number of nodes contained in each component and are re-rolled. The effect of the re-rolling is to SIMD apply the innermost loop. The eight iterations of the surrounding loop are automatically unrolled. Additionally, the outer loop over components is subjected to SWPL.

The results of this SIMD apply via loop re-rolling are discussed in the next section.

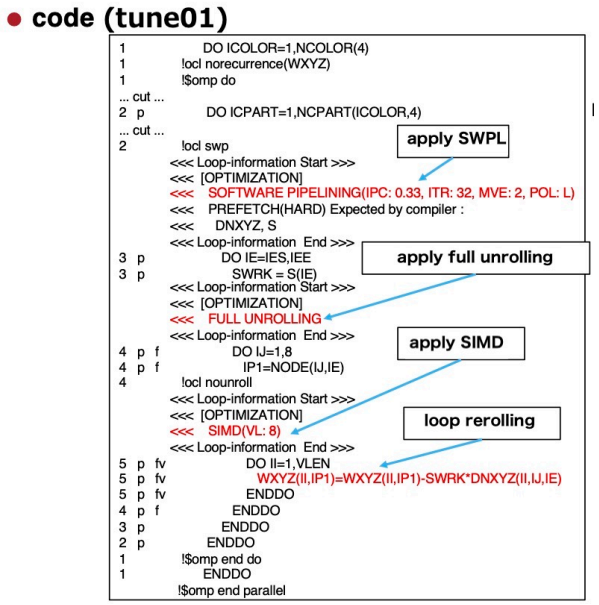


Figure 5.41: Loop structure of the tuned code.

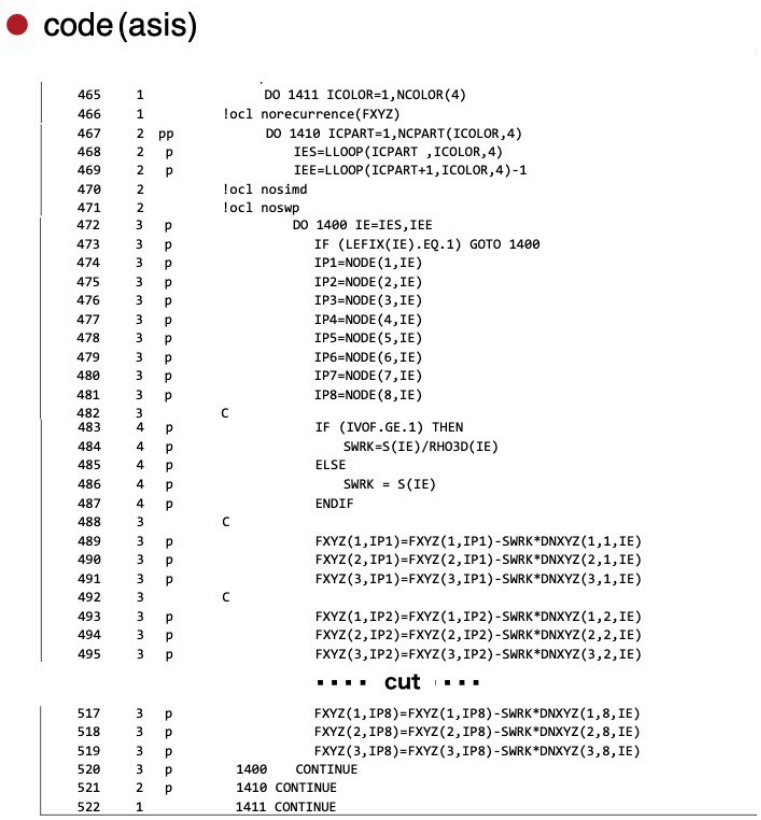


Figure 5.40: Loop structure of the original code.

5.3.3 Reduce indirect stores - converting from element loops to node loops

We first explain the original loop structure, referring to the diagram of Figure 5.42 and the code listing of Figure 5.43. In this code, the indices of nodes for which values are stored are accessed via lists, so `scatter store` instructions are used. The penalty incurred by the use of these instructions was not significant on the K computer due to its narrow SIMD width (two), but is significant for Fugaku (SIMD width eight), thus suggesting a degradation in execution efficiency.

- Use a loop structure in which we iterate over elements
- Values must be stored for each node
- Indices of nodes for which values are stored are accessed via lists
- Store-side list accesses are slow and should be avoided whenever possible
- Stores involve recurrence, requiring coloring to eliminate recurrence

Next, we will explain the tuning steps used to improve the original loop structure, referring to the diagram of Figure 5.44 and the code listing of Figure 5.45. The tuning involves restructuring loops to iterate over nodes instead of iterating over elements. This has the effect of avoiding performance-degrading `scatter-store` instructions, thus allowing efficient contiguous access for storage operations.

- Loop restructured to iterate over nodes rather than iterating over elements
- Relevant values for neighboring nodes are accumulated
- Indices of nodes for which values are stored yield contiguous accesses
- No need for coloring

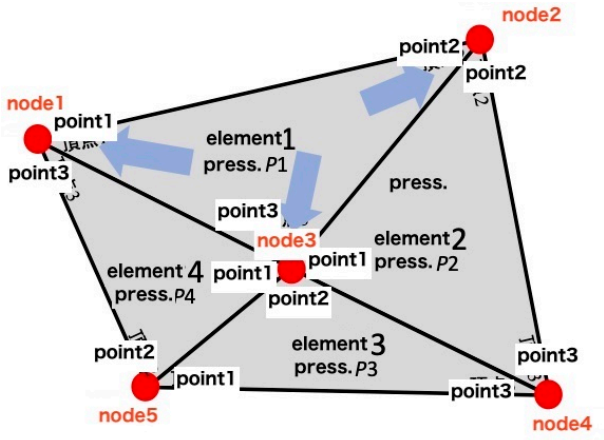


Figure 5.42: Schematic depiction of original procedure (asis)

asis

```
1              DO 1411 ICOLOR=1,NCOLOR(4)
1           !ocl norecurrence(FXYZ)
1           !$omp do
2  p           DO 1410 ICPART=1,NCPART(ICOLOR,4)
...cut...
2           !ocl nosimd
2           !ocl swp
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      LEFIX
          <<<    SPILLS :
          <<<      GENERAL   : SPILL 0  FILL 3
          <<<      SIMD&FP   : SPILL 0  FILL 0
          <<<      SCALABLE  : SPILL 0  FILL 0
          <<<      PREDICATE : SPILL 0  FILL 0
          <<< Loop-information  End >>>
3  p              DO 1400 IE=IES,IEE
3  p                IF (LEFIX(IE).EQ.1) GOTO 1400
3  p                IP1=NODE(1,IE)
 ...cut...           . . . . . .
3  p                IP8=NODE(8,IE)
 ...cut...           . . . . . .
3  p                FXYZ(1,IP1)=FXYZ(1,IP1)-SWRK*DNXYZ(1,1,IE)
3  p                FXYZ(2,IP1)=FXYZ(2,IP1)-SWRK*DNXYZ(2,1,IE)
3  p                FXYZ(3,IP1)=FXYZ(3,IP1)-SWRK*DNXYZ(3,1,IE)
 ...cut...           . . . . . .
3  p                FXYZ(1,IP8)=FXYZ(1,IP8)-SWRK*DNXYZ(1,8,IE)
3  p                FXYZ(2,IP8)=FXYZ(2,IP8)-SWRK*DNXYZ(2,8,IE)
3  p                FXYZ(3,IP8)=FXYZ(3,IP8)-SWRK*DNXYZ(3,8,IE)
3  p          1400   CONTINUE
2  p          1410 CONTINUE
1           !$omp end do
1            1411 CONTINUE
           !$omp end parallel
```

use array of NODE

storing point numbers
using index list

Figure 5.43: Source code for original procedure (asis)

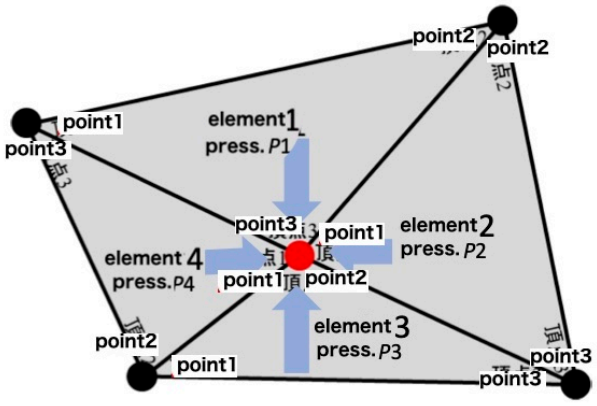


Figure 5.44: Schematic depiction of tuned procedure (tune02)

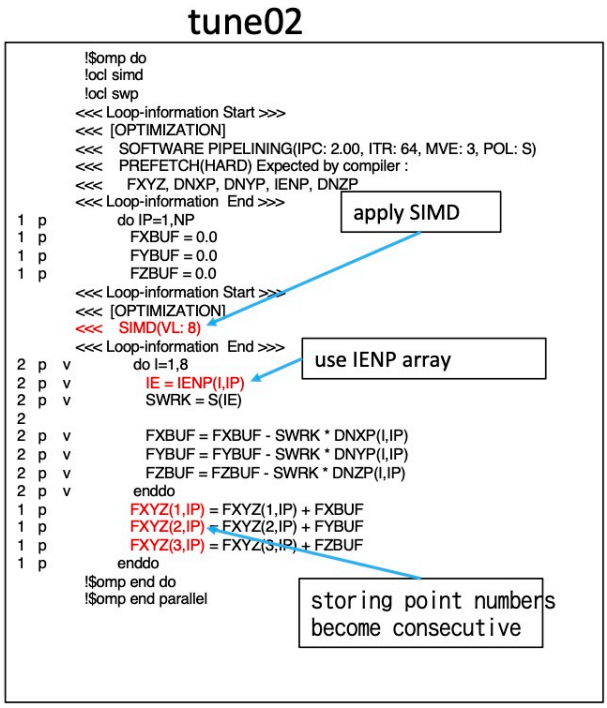


Figure 5.45: Source code for tuned procedure (tune02)

The performance effect of the tuning step described in this section is displayed graphically in Figure 5.46 and tabulated quantitatively in Figure 5.47. `tune01` refers to the partially-tuned code produced by SIMD-ed via loop re-rolling. `tune02` refers to the fully-tuned code involving both SIMD-ed via loop re-rolling and loop restructuring from elements to nodes.

`tune01` achieves the desired effect of significantly reducing the execution count via SIMD. However, the effect of indirect data access remains significant, and the DM rate for the L1D cache misses increases as prefetching fails to achieve its objective. Wait times for memory and cache access also contribute prominently. Consequently, the first partial tuning step makes only a minimal contribution to improving overall execution time, but yields a large reduction in instruction count that significantly decreases the L1D cache and functional unit busy times, thus making it a useful precursor for the next tuning step. The node reordering step in `tune02` dramatically reduces indirect access in storage operations, significantly reducing L1D-cache busy time and greatly enhancing memory throughput to yield a major performance improvement. The execution time of the fully tuned program then approaches the L2-cache busy time, thereby yielding performance close to theoretical limits.

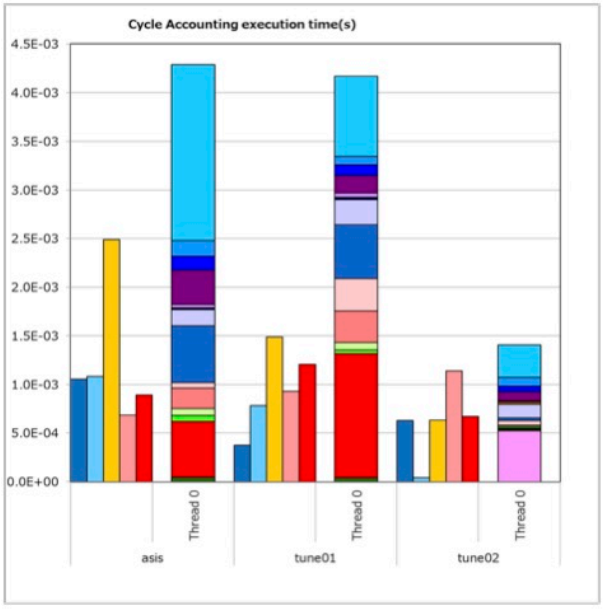


Figure 5.46: Performance charts showing the effect of loop restructuring from elements to nodes (tune02)

source code version	asis	tune01	tune02
tuning contents	–	apply SIMD - loop reroling -	reduce list access
compiler version	tcsds-1.2.27b	tcsds-1.2.27b	tcsds-1.2.27b
frequency(GHz)	2.0	2.0	2.0
kernel coded	yes	yes	yes
floating point precision	single	single	single
gathering thread #	0	0	0
elapsed time[s]	4.24E-03	4.12E-03	1.38E-03
GFLOPS/process	12.02	65.96	192.80
memory throughput(R+W) GB/s/process	54.04	75.14	124.59
float. point ops/thread	4.80E+06	2.56E+07	2.21E+07
executed instructions/thread	1.69E+07	7.70E+06	3.62E+06
load/store instructions/thread	8.83E+06	3.58E+06	1.06E+06
L1D misses/thread	7.57E+04	9.98E+04	5.07E+04
L2 misses/thread	6.84E+04	8.38E+04	5.04E+04
L1 busy ratio	58.71%	36.12%	45.71%
L2 busy ratio	16.10%	22.63%	82.64%
memory busy ratio	21.11%	29.35%	48.67%

Figure 5.47: Performance data showing the effect of loop restructuring from elements to nodes (tune02)

5.3.4 Adding prefetch directives for indirect memory accesses

Prefetching typically yields minimal performance gains for indirect memory accesses, but can have a nonnegligible effect in the following situations:

- When a comparison of contiguous to non-contiguous (indirect) memory accesses reveals that the latter are more prevalent.
- When arithmetic operations are present to some extent, but indirect memory accesses are more numerous.
- When the indices of nodes for which values are to be stored are themselves accessed via list.
- When cache wait times are responsible for a significant portion of the execution time.
- When storage operations involve recurrence, the elimination of which requires coloring.

If conditions such as these are present, it can be possible to improve performance by inserting software-prefetch optimization directives into the source code. These directives improve performance by preloading into the L1 cache the values that will be needed on the following loop iteration.

Asis (untuned) and partially tuned versions of a program demonstrating this technique are shown in Figure 5.48. The partially tuned program (`tune01`) specifies that eight values from the list-accessed `FXYZ` array are to be prefetched on each loop iteration.

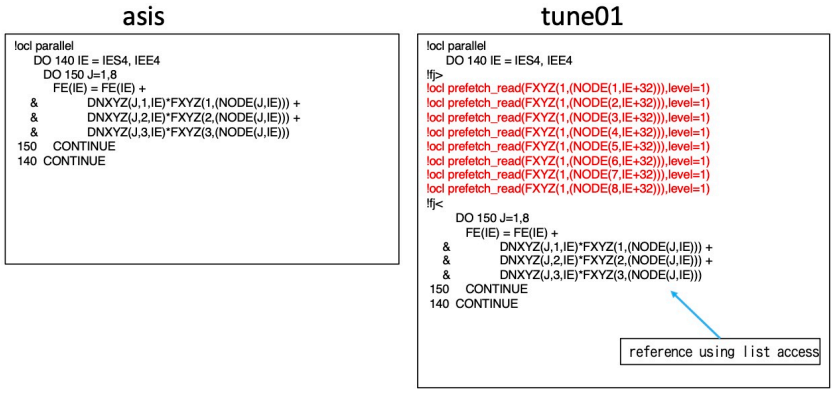


Figure 5.48: Asis (left) and partially-tuned (right) versions showing the use of prefetch directives

Further tuning to improve efficiency via prefetching yields the final fully-tuned program `tune02` of Figure 5.49. Whereas `tune01` invokes eight scalar prefetch instructions and non-SIMD loads to access list indices in the `NODE` array, `tune02` replaces the scalar prefetches with an efficient `gathering` prefetch instruction and uses SIMD loading to access list indices, thereby yielding further improvements in efficiency.

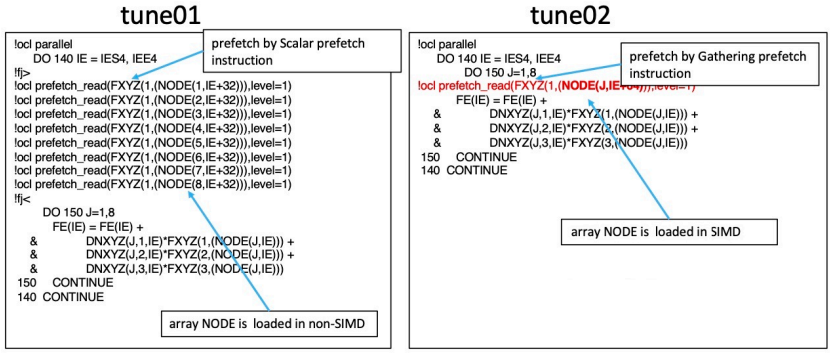


Figure 5.49: Fully-tuned version (tune02)

The performance effect of adding prefetch directives is displayed graphically in Figure 5.50 and tabulated quantitatively in Figure 5.51. The partially-tuned `tune01` program reduces the execution time from 10.6 to 8.83 seconds, but the addition of eight prefetch directives increases the instruction count. In the fully-tuned `tune02` program, the SIMD-ed loads from the `NODE` array determining the locations of prefetches, and the use of `gathering` prefetching for prefetch instructions, reduces the load/store instruction count. `tune02` also increases the SIMD instruction ratio from 24.15% to 77.24% and reduces the execution time from 8.83 seconds (for `tune01`) to 6.18 seconds. Finally, `tune02` improves memory throughput to a level near its theoretical upper bound.

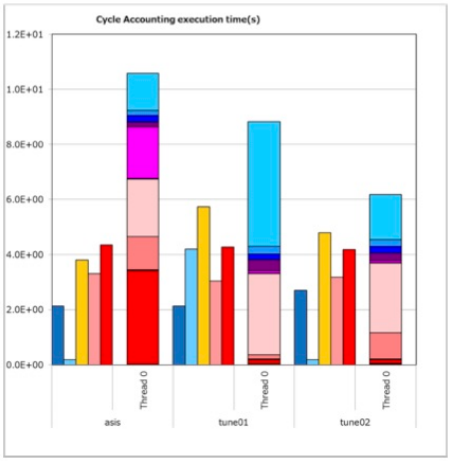


Figure 5.50: Performance charts showing the effect of prefetching directives for indirect memory accesses.

source code version	asis	tune01	tune02
tuning contents	-	add prefetch directives	modify prefetch directives
compiler version	tcsds-1.2.27b	tcsds-1.2.27b	tcsds-1.2.27b
frequency(GHz)	2.0	2.0	2.0
kernel coded	yes	yes	yes
floating point precision	single	single	single
gathering thread #	0	0	0
elapsed time[s]	1.06E+01	8.83E+00	6.18E+00
GFLOPS/process	77.67	93.00	132.96
memory throughput(R+W) GB/s/process	105.12	123.72	173.39
float. point ops/thread	6.84E+10	6.84E+10	6.84E+10
executed instructions/thread	1.31E+10	3.96E+10	1.61E+10
load/store instructions/thread	4.78E+09	9.03E+09	5.31E+09
L1D misses/thread	3.60E+08	3.48E+08	3.49E+08
L2 misses/thread	3.19E+08	3.16E+08	3.16E+08
L1 busy ratio	36.00%	64.83%	77.60%
L2 busy ratio	31.22%	34.45%	51.42%
memory busy ratio	41.06%	48.33%	67.73%
SIMD instruction ratio	72.91%	24.15%	77.24%
Gathering prefetch instr./thread	0.00E+00	0.00E+00	5.31E+08
Scalar prefetch instr./thread	4.17E+03	4.24E+09	1.75E+03

Figure 5.51: Performance data showing the effect of prefetching directives for indirect memory accesses.

5.4 Tuning Technique 3: Assorted optimizations for complex programs

The tuning techniques discussed in this section are applicable to in-cache applications with low B/F ratios for which optimal tuning is difficult due to the complexity of the code (Figure 5.52).

As noted previously, in comparison to the K computer, Fugaku has fewer registers, higher latency for arithmetic instructions, and larger SIMD width. These factors make it more difficult to apply instruction scheduling, and thus even in-cache applications often fail to achieve optimal performance. In this section, we discuss tuning methods that have succeeded in improving performance—if only slightly—for applications in this category.

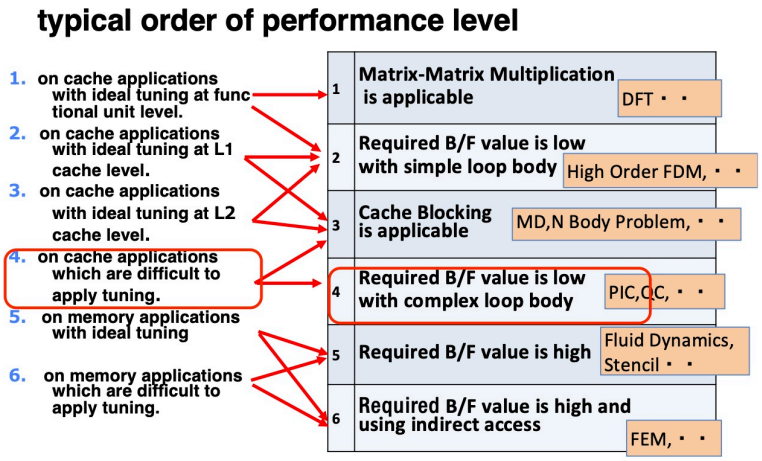


Figure 5.52: Application categories to which Tuning Technique 3 is applicable.

5.4.1 Expanding SIMD length

This technique is shown by the code listed in Figure 5.53. In this code, the innermost loop has been SIMD-ed. However, because `NB=4`, only four lanes are active, with the remaining 12 lanes deactivated by the predicate. We also note that the second index of the `rg` array ranges over the values 1-3.

```
do ie=1,NL/NB
   iee=ieo+ie-1
   do ivec=1,NB
   iei=ivec+(ie-1)*NB
   t1 = rg(ivec,1,cny(1,iee))
   t2 = rg(ivec,2,cny(1,iee))
   t3 = rg(ivec,3,cny(1,iee))
   t4 = rg(ivec,1,cny(2,iee))
      . . . . .
   t15= rg(ivec,3,cny(5,iee))

   rg(ivec,1,cny(1,iee))=t1+BDBuX(iei,1)
   rg(ivec,2,cny(1,iee))=t2+BDBuX(iei,2)
   rg(ivec,3,cny(1,iee))=t3+BDBuX(iei,3)
   rg(ivec,1,cny(2,iee))=t4+BDBuX(iei,4)
       . . . . .
   rg(ivec,3,cny(5,iee))=t15+BDBuX(iei,15)
       . . . . .
   enddo
 enddo ! ie
```

Figure 5.53: Asis version for SIMD expansion.

Our tuned version of this program is shown in Figure 5.54. By increasing `ivec` from four to `NB`×3=12, we access 12 contiguous components, thereby activating 12 SIMD lanes. In FORTRAN we could achieve the same effect by changing accesses to the form `rg(ivec,1,·,·)`.

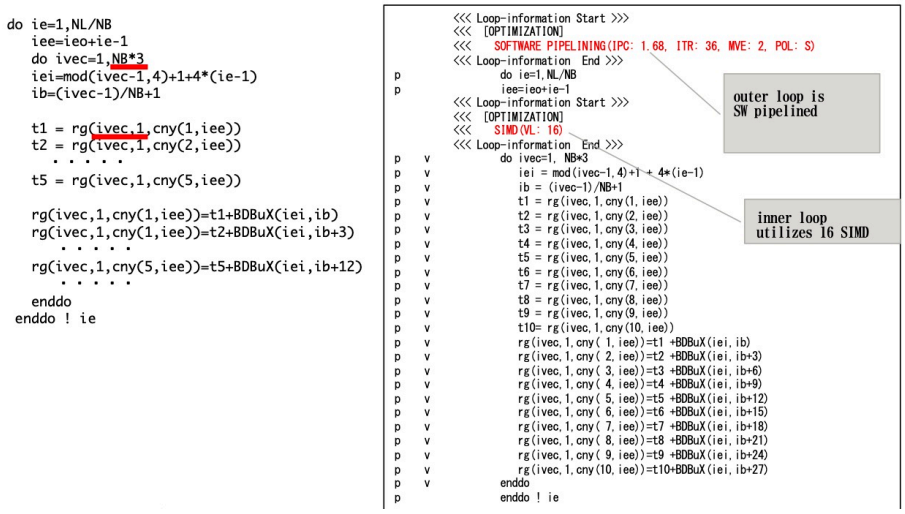


Figure 5.54: Tuned version of SIMD expansion.

The performance effect of this tuning step is displayed graphically in Figure 5.55 and tabulated quantitatively in Figure 5.56. The execution time is improved by around 7%. The reduction in the number of instructions executed—particularly load/store instructions—reduces instruction commit time. Additionally, software pipelining is applied to the outer loop, thereby improving instruction scheduling.

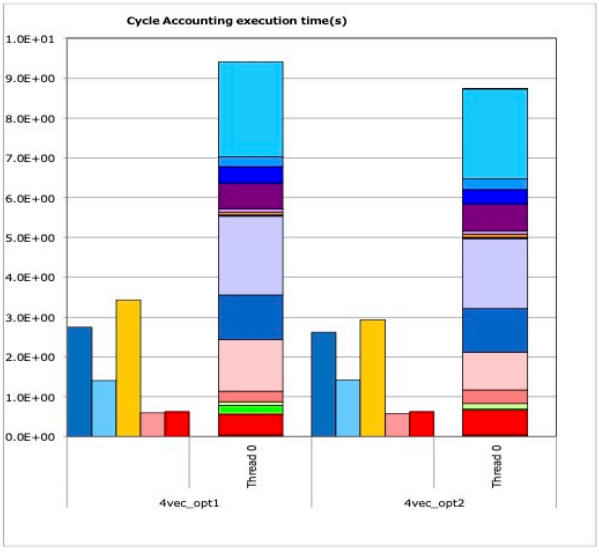


Figure 5.55: Graphical representation of performance benefit from SIMD expansion.

source code version	4vec_opt1	4vec_opt2
tuning contents	avoid excessive SFI (in list access)	extend SIMD length
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes/no	yes/no
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	9.41E+00	8.74E+00
GFLOPS/process	187.95	194.48
memory throughput (R+W) GB/s/process	17.10	18.27
float. point ops/thread	1.54E+11	1.48E+11
executed instructions/thread	2.35E+10	2.26E+10
load/store instructions/thread	7.27E+09	6.44E+09
L1D misses/thread	6.86E+07	6.77E+07
L2 misses/thread	3.83E+07	3.79E+07
L1 busy ratio	38.39%	36.30%
L2 busy ratio	6.23%	6.60%
memory busy ratio	6.68%	7.14%
SFI ratio	0.12	0.03

Figure 5.56: Performance data with and without SIMD expansion.

5.4.2 Suppressing inter-loop optimization to reduce register use

When the length of an innermost loop is short with a compiler-determined iteration count, the loop can be eliminated at compile time in favor of SIMD instructions. In this case, registers are used to share common expressions and loaded variables between loops, and, in actual calculations, a shortage of registers can result in large numbers of spill/fill events, frequently degrading performance. A useful tuning technique for such cases is to suppress inter-loop optimization by enclosing each loop inside IF statements that use provisional parameters to evaluate conditions. This process is shown in Figure 5.57.

```
subroutine cal_amat_s_color(..., true)
logical true(15)
     :
!$OMP PARALLEL !
$OMP DO
      do iu=1, np
        do ieo=nabs, nabe, NL
            if(NL .le. nabe-ieo+1) then
            if(true(1)) then
              do ie=1, NL
                indirect access calculation
              enddo
            endif
            if(true(2)) then
              do ie=1, NL
                Bu*calculation
              enddo
            endif
                 :
```

Figure 5.57: Suppressing inter-loop optimization (`tune06`).

Suppressing inter-loop optimization can cause a single calculation to be performed multiple times, thereby increasing the number of instructions executed. However, the use of fewer registers and the reduced occurrence of register spills have the effect of improving instruction scheduling while also reducing functional unit wait times and cache-access wait times, thus raising overall performance. For recent compiler versions, the `OCL opt_barrier` directive can be used instead of adding explicit IF statements.

The effect of this tuning step is depicted graphically in Figure 5.58, in which the left, center, and right performance charts are for the original (untuned) program, the program tuned by adding explicit IF statements, and the program tuned by adding the `OCL opt_barrier` directive, respectively. Note that both tuning approaches yield nearly identical performance levels.

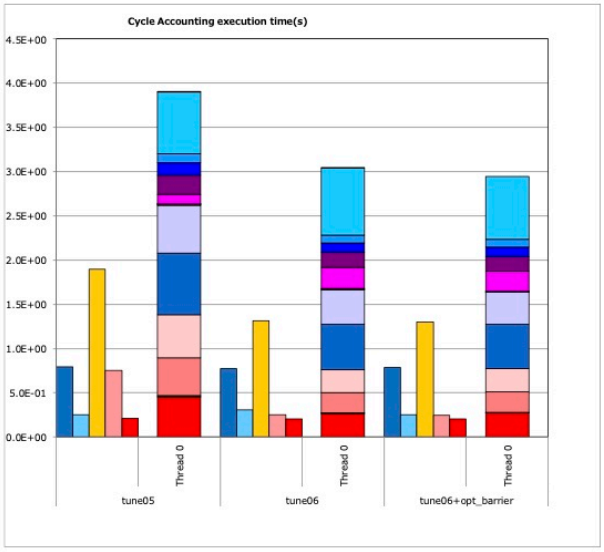


Figure 5.58: Performance chart of suppressing inter-loop optimization (`tune06`).

5.4.3 Improving instruction scheduling

Next, let's consider a program that loads a value from an array using list-based indirect indexing, performs some computation, and stores the result in the same array, with this procedure corresponding to several lines of source code. An example is shown in Figure 5.59.

In cases like this, the compiler determines, at compile time, that multiple indirect array element accesses from different lines of source code can involve the same address, in which case data storage on iteration i can come after data loading on iteration $i + 1$. This prohibits instruction scheduling.

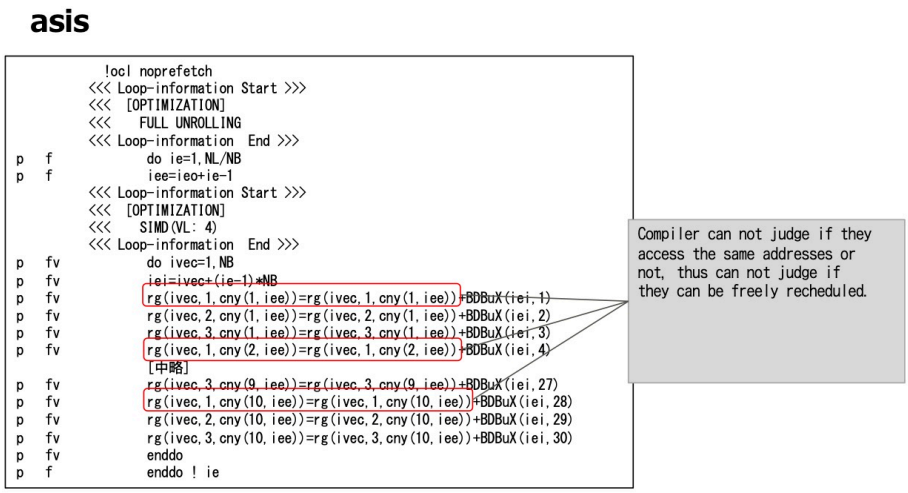


Figure 5.59: improving instruction scheduling (asis)

In cases like this, if list accesses are known not to overlap, load and store operations can be explicitly separated, as shown in Figure 5.60, to allow the compiler to invoke efficient scheduling.

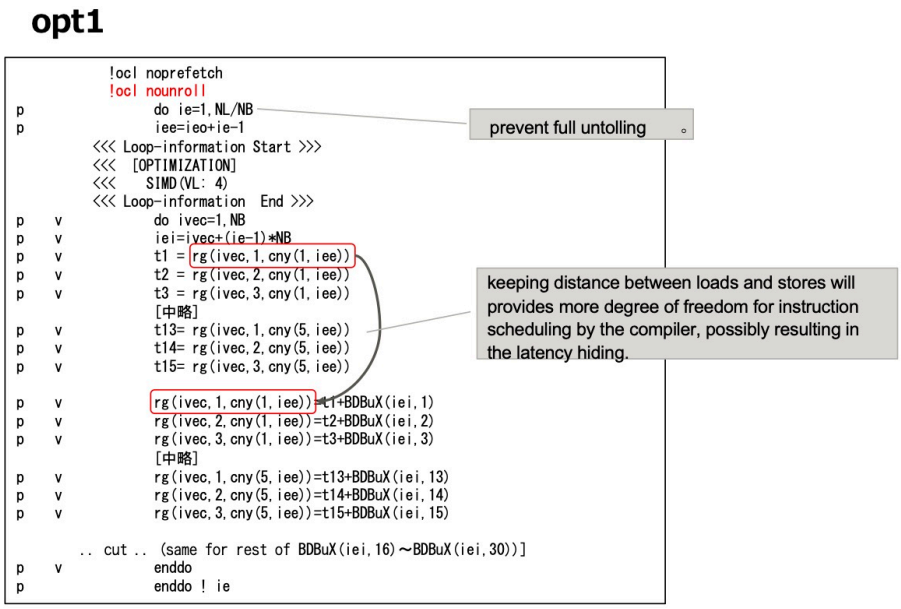


Figure 5.60: improving instruction scheduling (`opt1`)

The performance effect of this tuning step is displayed graphically in Figure 5.61 and tabulated quantitatively in Figure 5.62. The numbers of executed instructions and of load/store instructions both decrease, thus reducing program runtime by around 17%. The difference between the untuned and tuned codes shows the effect of instruction scheduling. By explicitly invoking advance data loading from array `rg` in the source code, we modify the scheduling of instructions, thereby ensuring adequate separation between array accesses. The reduction in load/store instructions comes together with a reduction in load instructions for array `cny` (as a side effect).

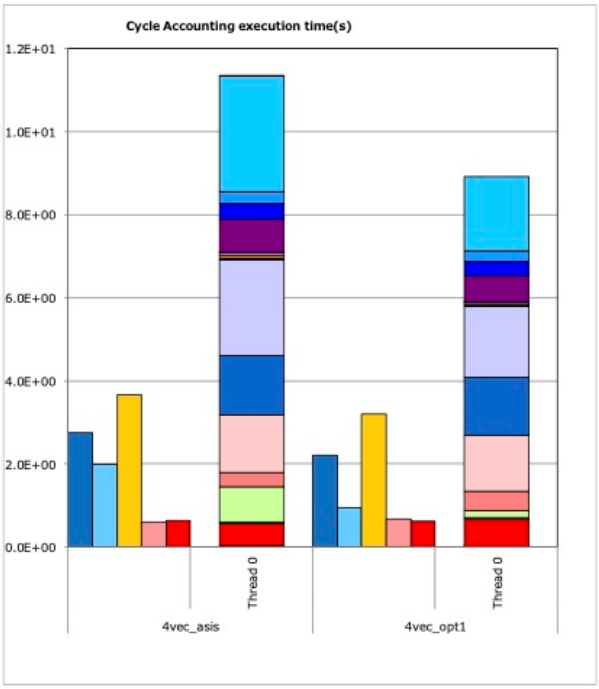


Figure 5.61: Performance charts showing the effect of improved instruction scheduling.

source code version	asis	opt1
tuning contents		load/store separation
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes/no	yes/no
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	1.13E+01	9.41E+00
GFLOPS/process	155.97	187.95
memory throughput (R+W) GB/s/process	14.20	17.10
float. point ops/thread	1.54E+11	1.54E+11
executed instructions/thread	2.70E+10	2.35E+10
load/store instructions/thread	7.96E+09	7.27E+09
L1D misses/thread	6.76E+07	6.86E+07
L2 misses/thread	3.84E+07	3.83E+07
L1 busy ratio	34.30%	38.39%
L2 busy ratio	5.33%	6.23%
memory busy ratio	5.55%	6.68%
SFI ratio	0.11	0.12

Figure 5.62: Performance data showing the effect of improved instruction scheduling.

5.4.4 Reducing the number of instructions executed

Using the Fugaku compiler to translate kernel programs increases the prevalence of `mov` and `add` instructions for integer arithmetic. These instructions are used as follows to access arrays or other entities in memory.

`x1` ← `5600`	Load explicit value 5600 into register `x1` (generates `mov` instruction)
`x2` ← `sp + x1`	Set `x2` to `sp+5600` (generates `add` instruction)
`z1` ← `[x2]`	Load data from address stored in `x2`

The `add` instructions allows explicit operands in the range [-4096,+4095]. Addresses separated from the stack pointer (`sp`) by more than 4096 bytes must be computed by using a `mov` instruction to load an offset into a register. No `mov` instruction is required for addresses near the stack pointer, so the number of `mov` instructions can be reduced by positioning frequently-accessed arrays at nearby locations. In this section, we discuss tuning procedures for reducing `mov` instructions.

To simplify this discussion, before explaining the tuning method itself, we will first explain the actual tuning procedure carried out in this case. Figure 5.63 shows the tuning process that converts `tune06` into `tune07`. Here, we are rearranging computations as indicated in Figure 5.63. The objectives of this rearrangement include sharing common expressions, recasting arithmetic operations as accumulations, and ensuring adequate separation between stores to the same variable.

Figure 5.63: The tuning process from `tune06` to `tune07`.

Next, we will explain the tuning procedure for reducing `mov` instructions. Private variables in OpenMP are stored on the stack and can be assumed to be ordered in the sequence specified in `private` sections, so we can reduce `mov` instructions by declaring `private` variables in decreasing order of access frequency, as shown in Figure 5.64.

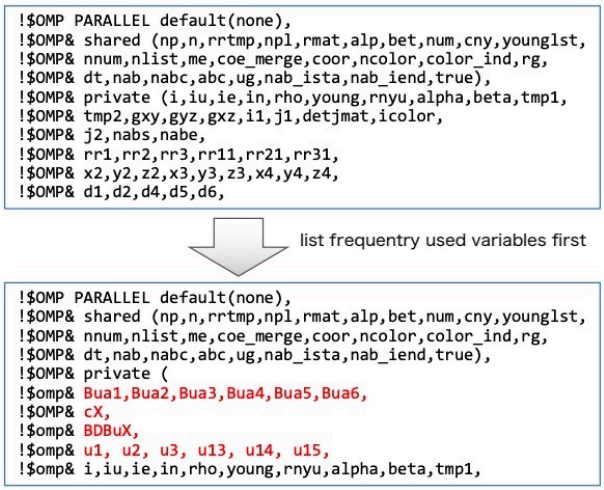


Figure 5.64: Tuning the order in which `private` variables are declared.

Additionally, the load instructions used to access arrays allow the specification of offsets for each SIMD width, so we can reduce `mov` and `add` instructions by aggregating arrays, as shown in Figure 5.65. Applying this tuning step to `tune07` yields `tune08`.

The performance effect of reducing executed instructions is displayed graphically in Figure 5.66 and tabulated quantitatively in Figure 5.67. The instruction count is significantly reduced when transitioning from `tune07` to `tune08`, which demonstrates the benefit of eliminating `mov` instructions. Execution time is also reduced by 13%.

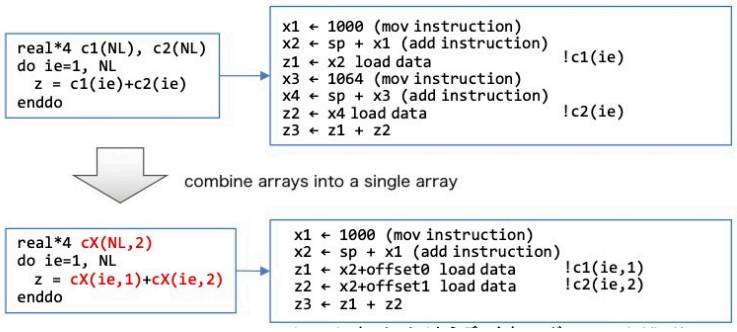


Figure 5.65: Aggregating arrays to reduce executed instructions.

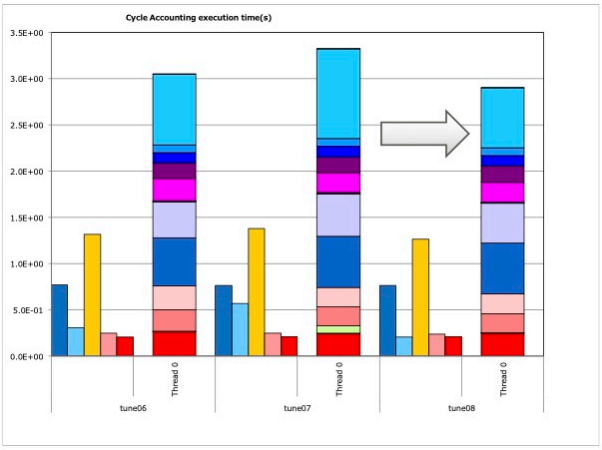


Figure 5.66: Performance charts showing the effect of reducing executed instructions.

source code version	tune06	tune07	tune08
tuning contents	suppress inter-loop optimization	change instruction scheduling	reduce executed # of instructions
compier version	tcsu-1.2.26	tcsu-1.2.26	tcsu-1.2.26
frequency(GHz)	2.0	2.0	2.0
kernel coded	yes/no	yes/no	yes/no
float. point precision	single	single	single
gathering thread #	0	0	0
elapsed time[s]	3.05E+00	3.32E+00	2.90E+00
GFLOPS/process	129.47	129.01	147.68
memory throughput (R+W) GB/s/process	17.48	16.45	18.73
float. point ops/thread	3.43E+10	3.73E+10	3.73E+10
executed instructions/thread	7.36E+09	9.06E+09	6.50E+09
load/store instructions/thread	2.84E+09	2.87E+09	2.78E+09
L1D misses/thread	3.10E+07	2.93E+07	2.96E+07
L2 misses/thread	1.32E+07	1.34E+07	1.34E+07
L1 busy ratio	45.01%	43.35%	45.38%
L2 busy ratio	8.21%	7.47%	8.25%
memory busy ratio	6.83%	6.43%	7.32%
other instructions/thread	2.24E+09	3.60E+09	1.43E+09

Figure 5.67: Performance data showing the effect of reducing executed instructions.

5.4.5 Using enhanced compiler optimization features

Improvements in compiler optimization capabilities now allow SIMD processing of leftover loop portions resulting from SIMD loops with lengths shorter than the SIMD width of the machine. This feature is enabled by default, so the “original” (untuned) version of the program, in this case, is obtained by adding a compiler directive to

disable it. We refer to the code version without this compiler directive as `tune1`. The effect of this tuning step is discussed in Section 5.4.7 below.

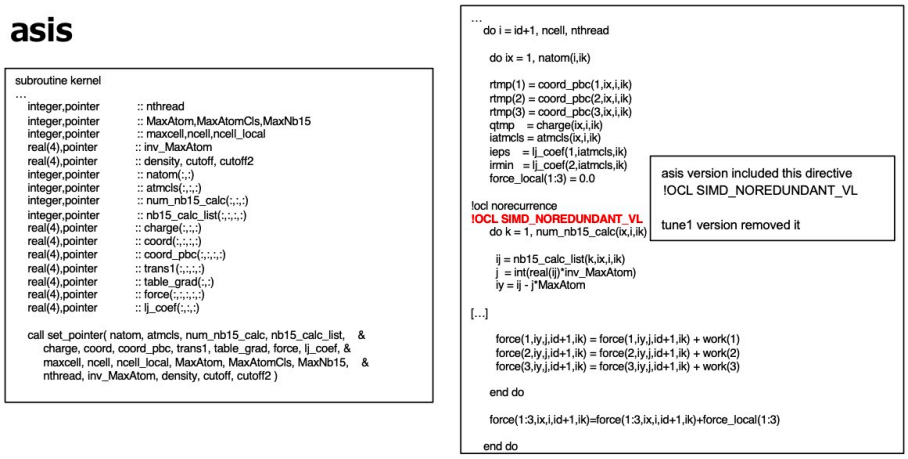


Figure 5.68: apply SIMD to residual loop portions (original, untuned code).

5.4.6 supplement compiler optimization - use explicit arguments instead of pointer variables

The use of data structures containing pointer types can obstruct the compiler optimization process. To avoid this, subroutines accepting array-valued arguments are rewritten with additional new arguments specifying array dimensions, as shown in Figure 5.69. In some cases, this can allow compiler optimization to proceed, thereby yielding improved performance. Array declarations appear on the receiving side. Applying this tuning step together with the tuning step discussed in Section 5.3.4 (adding prefetch directives to codes using list-accessed arrays) yields a new program version named `tune2`. The effect of this tuning step is discussed in Section 5.4.7 below.

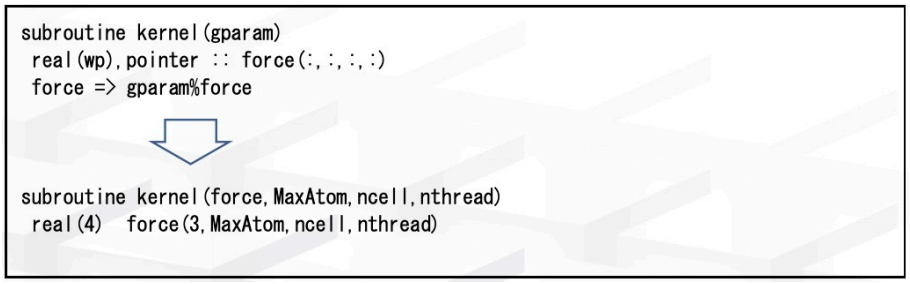


Figure 5.69: supplement compiler optimization - use explicit arguments instead of pointer variables

5.4.7 supplement compiler optimization - adding `contiguous` attribute to array of pointers

In the previous section, we noted that converting pointer variables to subroutine parameters can improve performance in some cases. However, carrying out this step for real-world applications can involve a significant amount of effort because the program portions that require revision can be rather extensive. If the contents of the data structures in question are stored contiguously in memory and do not overlap other regions, pointer variables can be declared with the `contiguous` attribute to indicate that they refer to contiguous memory regions. This is demonstrated by the program shown in Figure 5.70. Specifying the `contiguous` attribute allows compiler optimization to proceed on the assumption that the memory region in question is contiguous, thus achieving the same effect as the specification of subroutine parameters. Applying this tuning technique yields the code labeled `tune3`.

tune3

```
subroutine kernel
...
  integer,pointer           :: nthread
  integer,pointer           :: MaxAtom,MaxAtomCls,MaxNb15
  integer,pointer           :: maxcell,ncell,ncell_local
  real(4),pointer           :: inv_MaxAtom
  real(4),pointer           :: density, cutoff, cutoff2
  integer,pointer,contiguous :: natom(:,:)
  integer,pointer,contiguous :: atmcls(:,:,:)
  integer,pointer,contiguous :: num_nb15_calc(:,:,:)
  integer,pointer,contiguous :: nb15_calc_list(:,:,:,:)
  real(4),pointer,contiguous :: charge(:,:,:)
  real(4),pointer,contiguous :: coord(:,:,:,:)
  real(4),pointer,contiguous :: coord_pbc(:,:,:,:)
  real(4),pointer,contiguous :: trans1(:,:,:,:)
  real(4),pointer,contiguous :: table_grad(:,:)
  real(4),pointer,contiguous :: force(:,:,:,:,:)
  real(4),pointer,contiguous :: lj_coef(:,:,:)

  call set_pointer( natom, atmcls, num_nb15_calc, nb15_calc_list,   &
      charge, coord, coord_pbc, trans1, table_grad, force, lj_coef, &
      maxcell, ncell, ncell_local, MaxAtom, MaxAtomCls, MaxNb15,    &
      nthread, inv_MaxAtom, density, cutoff, cutoff2 )
```

Figure 5.70: Adding `contiguous` attribute.

The performance effect of the tuning techniques discussed in this section and the previous two sections is shown graphically in Figure 5.71 and tabulated quantitatively in Figure 5.72. Applying the technique discussed Section 5.4.5 to the original (untuned) code yields the partially-tuned code `tune1`, which reduces the number of instructions executed from $1.02{\cdot}10^7$ to $8.11{\cdot}10^6$ because the application of SIMD improves performance by a factor of 1.06. Furthermore, applying the technique of Section 5.4.6 yields the partially-tuned code `tune2`, in which the issuance of prefetches increases the number of instructions executed but slashes the L1D demand-miss rate from 87.13% to 20.95%, thereby improving performance by a factor of 1.33. Finally, applying the technique discussed in Section 5.4.7 yields our final, fully-tuned code `tune3`. The performance of `tune3` is essentially equivalent to that of `tune2`, but the tuning step required to produce `tune3` is very simple—requiring only that certain pointer variables be assigned the `contiguous` attribute—and is less labor-intensive than the process of rewriting the subroutine calling conventions required to produce `tune2`.

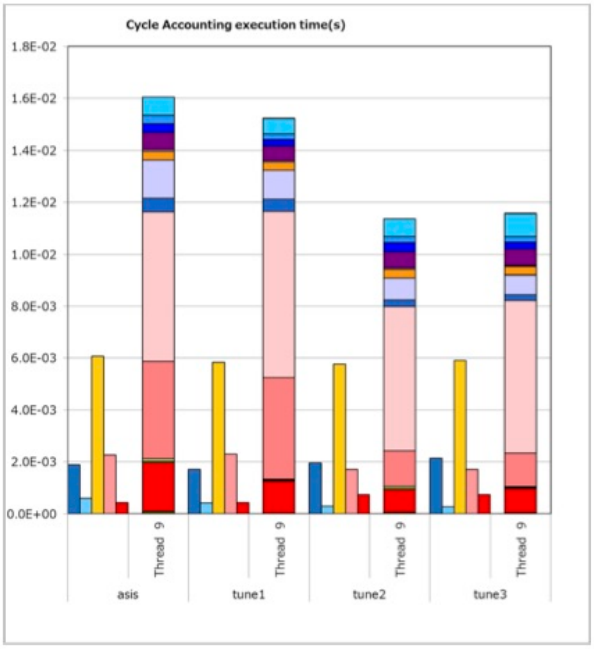


Figure 5.71: Performance charts showing the effect of the tuning techniques discussed in subsections 5.4.5, 5.4.6, and 5.4.7.

source code version	asis	tune1	tune2	tune3
tuning contents	–	- enhanced compiler optimization	– prefetch - use basic arrays instead of pointers	– prefetch – contiguous attribute to an array of pointers
compiler version	tcsu1.2.26	tcsu1.2.26	tcsu1.2.26	tcsu1.2.26
frequency(GHz)	2.0	2.0	2.0	2.0
kernel coded	yes	yes	yes	yes
float. point precision	single	single	single	single
gathering thread #	9	9	9	9
elapsed time[s]	1.61E−02	1.52E−02	1.14E−02	1.16E−02
GFLOPS/process	41.72	44.78	60.08	59.01
memory throughput (R+W) GB/s/process	6.89	7.24	16.54	16.28
float. point ops/thread	5.74E+07	5.85E+07	5.85E+07	5.85E+07
executed instr./thread	1.02E+07	8.22E+06	9.38E+06	1.08E+07
load/store instructions /thread	4.45E+06	3.54E+06	3.66E+06	4.49E+06
L1D misses/thread	2.64E+05	2.62E+05	2.43E+05	2.50E+05
L2 misses/thread	3.16E+04	3.10E+04	5.70E+04	5.71E+04
L1 busy ratio	37.82%	38.28%	51.02%	51.39%
L2 busy ratio	14.13%	15.12%	15.00%	14.85%
memory busy ratio	2.69%	2.83%	6.46%	6.36%
L1D miss dm ratio	87.19%	87.13%	20.95%	21.36%

Figure 5.72: Performance data showing the effect of the tuning techniques discussed in Sections 5.4.5, 5.4.6, and 5.4.7.

5.4.8 Parallelization via dynamic thread scheduling

When applications are parallelized via multithreading, the emergence of significant load imbalances between threads can degrade performance. In such cases, the method of dynamic thread scheduling, shown in Figure 5.73, can improve performance. In the original (untuned) version of Figure 5.73, the loop is parallelized by cyclic decomposition of the loop over `i`. The tuning step is simply to add the directive `!$omp do schedule(dynamic,1)` to enable dynamic thread scheduling.

```
!$omp do schedule(dynamic,1)
do i = 1, ncell
 do ix = 1, natom(i)
   rtmp(1) = coord_pbc(1,ix,i)
    :
   do k = 1, num_nb15_calc(ix,i)
```

Figure 5.73: Program demonstrating dynamic thread scheduling.

The performance effect of parallelization via dynamic thread scheduling is displayed graphically in Figure 5.74 and tabulated quantitatively in Figure 5.75. Compared to parallelization via simple cyclic splitting of the `i` loop in Figure 5.73, the dynamic approach achieves a 9.0% improvement in performance.

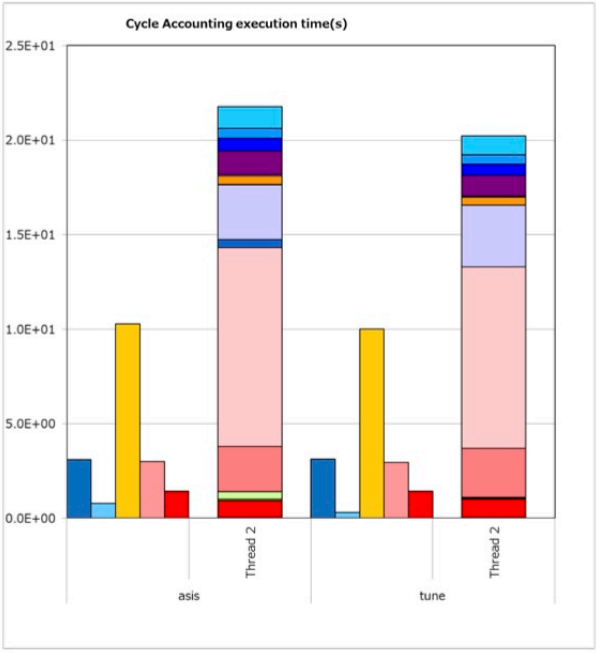


Figure 5.74: Performance charts showing the effect of dynamic thread scheduling.

thread	asis	tune
0	9.00E+10	8.16E+10
1	7.94E+10	8.12E+10
2	7.51E+10	8.15E+10

Figure 5.75: Performance data showing the effect of dynamic thread scheduling.

5.4.9 Eliminating array-index computations

This tuning method is shown by the original (untuned) program discussed in Figure 5.76. In this program, the index `ij` is obtained by using the loop counter `k` as an index into the `nb15_calc_list` array, after which the value of `ij` is used to compute values for indices `j` and `iy`, which are used to access the `force` array.

```
do i = 1,ncell
 do ix = 1, natom(i)
    :
   do k = 1, num_nb15_calc(ix,i)
      :
     ij = nb15_calc_list(k,ix,i) - MaxAtom
     j = int(real(ij)*inv_MaxAtom)
     iy = ij - j*MaxAtom
       :
     force(1,iy,j,id+1) = force(1,iy,j,id+1) + work(1)
     force(2,iy,j,id+1) = force(2,iy,j,id+1) + work(2)
     force(3,iy,j,id+1) = force(3,iy,j,id+1) + work(3)
```

Figure 5.76: Original (untuned) program demonstrating elimination of array-index computations.

We rewrite the code of Figure 5.76 as shown in Figure 5.77. This rewrite eliminates the calculations of indices `j` and `iy`, and uses only index `ij` to access the `force` array. In FORTRAN, the same effect can be achieved by writing `force(1,ij,1,id+1)`.

```
do i = 1,ncell
 do ix = 1, natom(i)
    :
   do k = 1, num_nb15_calc(ix,i)
      :
     ij = nb15_calc_list(k,ix,i) - MaxAtom
       :
     force(1,ij,1,id+1) = force(1,ij,1,id+1) + work(1)
     force(2,ij,1,id+1) = force(2,ij,1,id+1) + work(2)
     force(3,ij,1,id+1) = force(3,ij,1,id+1) + work(3)
```

Figure 5.77: The Figure 5.76 program after eliminating array-index computations (`tune`).

The performance effect of eliminating array-index calculations is displayed graphically in Figure 5.78 and tabulated quantitatively in Figure 5.79. These figures show that the performance of the tuned program is 1.11 times greater than that of the untuned program. The number of executed instructions is reduced from $2.44 \cdot 10^{10}$ to $1.97 \cdot 10^{10}$. Presumably, the eliminated instructions are used for array-index computations (integer arithmetic). The L1D cache-access wait time is reduced from 10.5 to 8.73 seconds, and instruction scheduling improves as well.

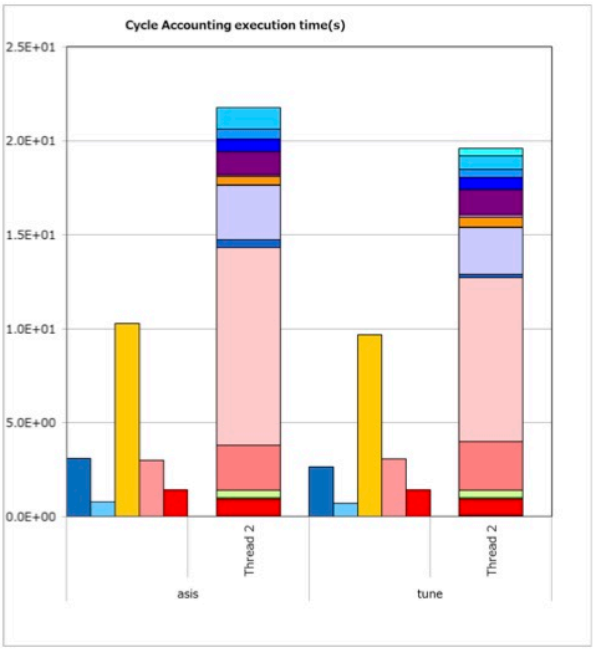


Figure 5.78: Performance charts showing the effect of eliminating array-index calculations.

source code version	asis	tune
tuning contents	–	remove array index calculation
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	2(rank#9)	2(rank#9)
elapsed time[s]	2.18E+01	1.96E+01
GFLOPS/process	46.71	49.54
memory throughput (R+W) GB/s/process	17.90	19.83
float. point ops/thread	7.51E+10	7.18E+10
executed instructions/thread	2.44E+10	1.97E+10
load/store instructions/thread	7.79E+09	5.60E+09
L1D misses/thread	4.14E+08	4.62E+08
L2 misses/thread	1.07E+08	1.07E+08
L1 busy ratio	51.28%	53.87%
L2 busy ratio	14.63%	16.61%
memory busy ratio	6.99%	7.75%

Figure 5.79: Performance data showing the effect of eliminating array-index calculations.

5.4.10 Disabling SIMD

Next, let's consider the program listed in Figure 5.80, and suppose the number of threads is `nthread`=3. Working in single-precision arithmetic, we have a SIMD width of 16. The code performs a reduction calculation, which—

on a 16SIMD machine—proceeds via the reduction pattern 16→8→4→2→1 and thus requires four computation stages. In some cases, applying SIMD can degrade performance by significantly increasing the number of arithmetic operations performed. In such cases, the `!ocl nosimd` compiler directive can easily be used to disable SIMD, as shown in Figure 5.80.

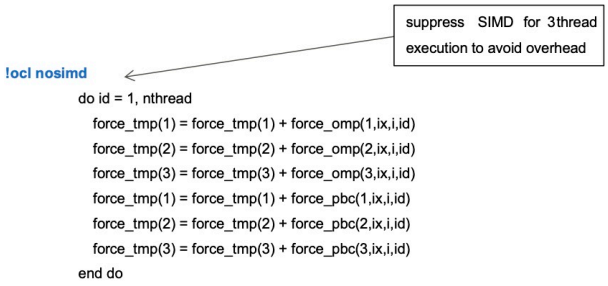


Figure 5.80: Program demonstrating the use of a compiler directive to disable SIMD.

The performance effect of disabling SIMD is displayed graphically in Figure 5.81 and tabulated quantitatively in Figure 5.82. From Figure 5.82, we can see that the number of floating-point arithmetic operations in the original (untuned) code is approximately 16 times greater than in the tuned code.

To understand this result, we first consider the number of arithmetic operations executed by the original code. `force_tmp(1-3)` is computed by summing `force_omp` and `force_pbc`, entailing (SIMD width 16)×(6 arithmetic operations)=96 operations. The number of operations performed in the SIMD-reduced calculation is 16×(4 stages)×(3 elements) = 192. Hence, the total number of arithmetic operations for the original code is 96+192=288. In contrast, the tuned code executes a total of (three iterations)×(6 operations)=18 operations. As a result, the ratio of arithmetic operations in the original and tuned codes is 288/18 = 16, as expected. In cases like this, disabling SIMD as in Figure 5.80 is a useful technique.

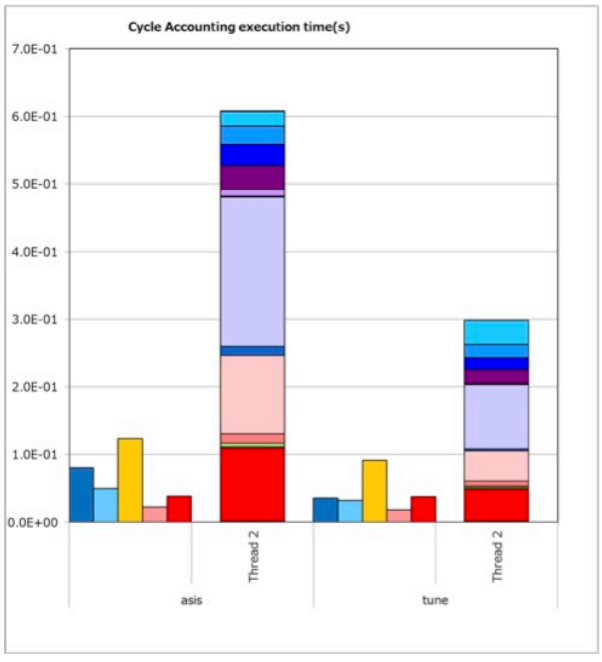


Figure 5.81: Performance charts showing the effect of disabling SIMD.

source code version	asis	tune
tuning contents	-	suppress SIMD
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	2(rank#9)	2(rank#9)
elapsed time[s]	6.08E-01	2.97E-01
GFLOPS/process	31.52	3.99
memory throughput (R+W) GB/s/process	16.18	32.51
float. point ops/thread	1.59E+09	9.86E+07
executed instructions/thread	5.34E+08	5.11E+08
load/store instructions/thread	1.30E+08	1.69E+08
L1D misses/thread	3.13E+06	3.07E+06
L2 misses/thread	2.70E+06	2.75E+06
L1 busy ratio	20.07%	30.70%
L2 busy ratio	3.68%	6.20%
memory busy ratio	6.32%	12.70%

Figure 5.82: Performance data showing the effect of disabling SIMD.

5.4.11 Rearrangement via ACLE

The process of rearrangement via ACLE is outlined in Figure 5.83. The original (untuned) code is shown at the left of this figure. This code is separated into computation and rearrangement portions, and the latter is rewritten using the Arm C Language Extensions (ACLE).

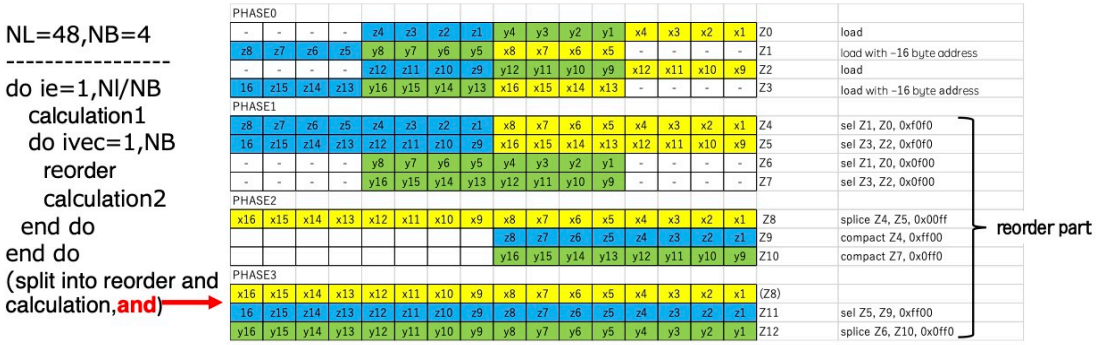


Figure 5.83: Overview of rearrangement via ACLE (`4vec_opt7`).

Rearrangement via ACLE is carried out via the following procedure, which is shown in Figure 5.83:

(1) Load first data element into register `Z0`.

(2) Load second data element, shifted by 16 bytes, into register `Z1`.

(3) Load third data element into register `Z2`.

(4) Load fourth data element, shifted by 16 bytes, into register `Z3`.

(5) Use a `SEL` instruction to extract x and z data from `Z0` and `Z1` and store the results in register `Z4`.

(6) Use a `SEL` instruction to extract x and z data from `Z2` and `Z3` and store the results in register `Z5`.

(7) Use a `SEL` instruction to extract y data from `Z0` and `Z1` and store the results in register `Z6`.

(8) Use a `SEL` instruction to extract y data from `Z2` and `Z3` and store the results in register `Z7`.

(9) Use a `SPLICE` instruction to combine the contents of `Z4` and `Z5`, then store the x data in `Z8`. (This completes the x rearrangement.)

(10) Use a `COMPACT` instruction to shift the z data for `Z4` to the front of the register and store it in `Z9`.

(11) Use a `COMPACT` instruction to shift the y data for `Z7` to the front of the register and store it in `Z10`.

(12) Use a `SEL` instruction to extract z data from `Z5` and `Z9` and store it in `Z11`. (This completes the z rearrangement.)

(13) Use a `SPLICE` instruction to combine the contents of `Z6` and `Z10`, then store the y data in `Z12`. (This completes the y rearrangement.)

The performance effect of rearranging via ACLE is displayed graphically in Figure 5.84 and tabulated quantitatively in Figure 5.85. Code regions that were not SIMD-ed in `4vec_opt3` are SIMD-ed in `4vec_opt7`. The number of instructions executed decreases, and the SIMD rate improves from 54% to 59%. Instruction scheduling is also improved, and functional unit wait time is reduced. The reduction in instructions executed reduces instruction commit time. Overall execution time improves by approximately 8%.

Note: ACLE is available for C/C++ in CLANG mode.

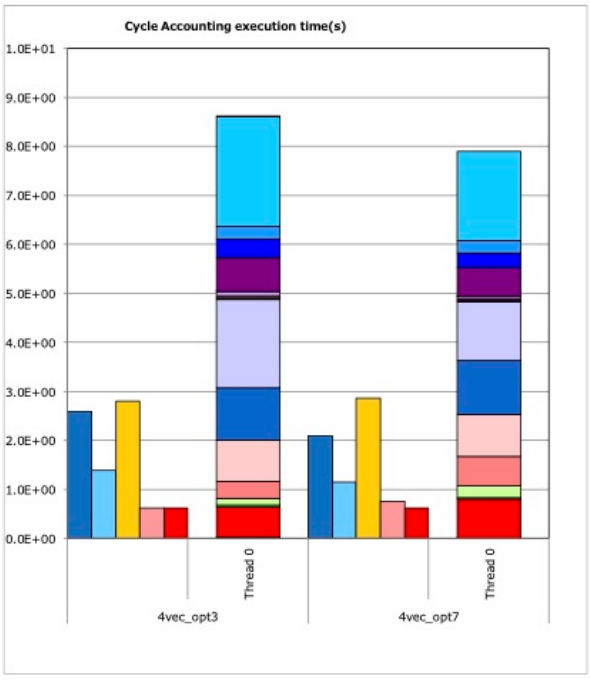


Figure 5.84: Performance charts showing the effect of rearranging via ACLE.

source code version	4vec_opt3	4vec_opt7
tuning contents	consecutive data access	data reordering using ACLE
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes/no	yes/no
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	8.60E+00	7.90E+00
GFLOPS/process	197.47	211.44
memory throughput (R+W) GB/s/process	18.55	20.18
float. point ops/thread	1.48E+11	1.45E+11
executed instructions/thread	2.24E+10	1.85E+10
load/store instructions/thread	6.40E+09	5.28E+09
L1D misses/thread	6.78E+07	8.72E+07
L2 misses/thread	3.82E+07	3.80E+07
L1 busy ratio	35.22%	37.16%
L2 busy ratio	7.17%	9.56%
memory busy ratio	7.24%	7.88%

Figure 5.85: Performance data showing the effect of rearranging via ACLE.

5.4.12 Manual scheduling

The manual scheduling technique is depicted schematically in Figure 5.86.

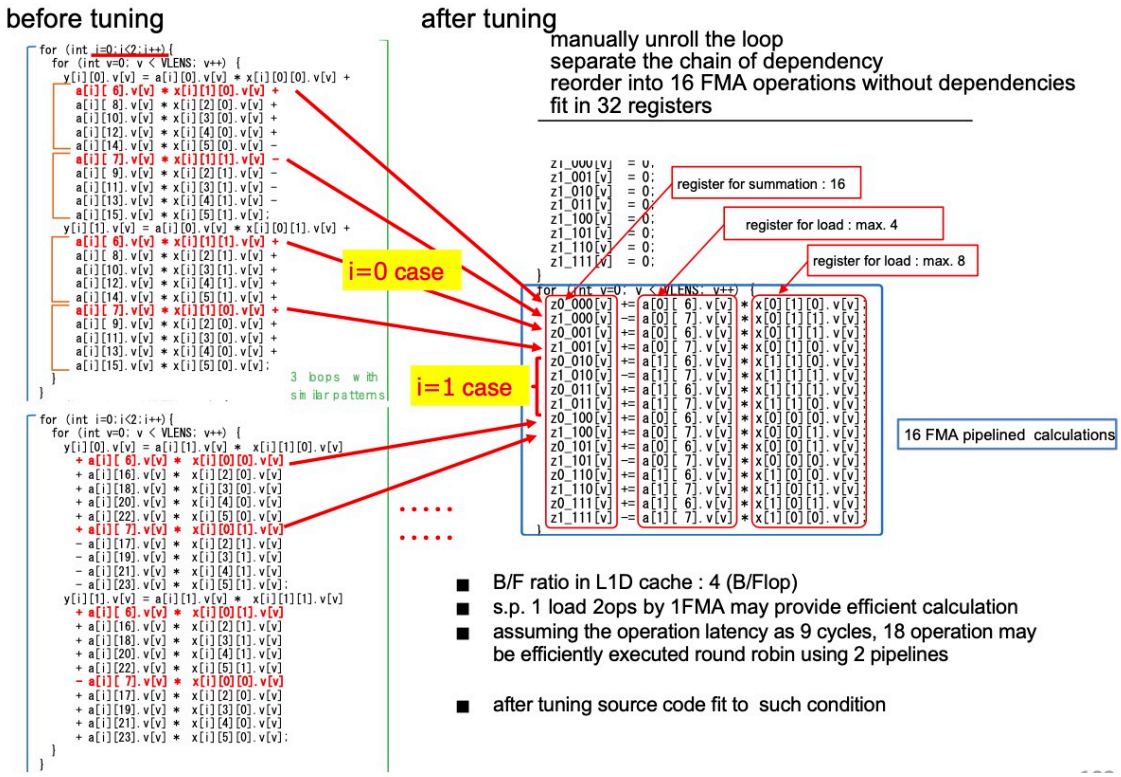


Figure 5.86: Schematic diagram showing manual scheduling.

The original (untuned) program demonstrating this technique is listed at the left of Figure 5.86. We now proceed to apply manual scheduling to this program, based on the following reasoning:

(1) The B/F ratio of the L1D cache is 4 bytes/flops.

(2) Thus, we expect that a calculation with a B/F ratio of one single-precision load to one FMA (4 bytes / 2 flops) should run in the L1D cache, and should thus achieve high performance.

(3) Assuming we have two instruction pipelines and an arithmetic latency of nine cycles, we should be able to achieve high performance by executing 18 computational chains in a round-robin configuration.

(4) However, it must be possible to execute the full set of calculations envisioned in item (3) without mutual interdependencies.

(5) In the original program shown on the left-hand side of Figure 5.86, the arithmetic operations indicated by red arrows satisfy the condition of Item (4), which means the program can be rewritten to execute these operations in the same loop.

(6) The upper section in the original code runs over the two values `i=0` and `i=1`, with eight lines of code executed for each value.

(7) Combining this with the lower section of the original code yields 16 lines of code.

(8) These 16 lines of code execute 32 separate 16-FMA operations.

(9) 16 registers are needed for storage, and a maximum of 12 registers are needed for loading, this indicating a total of 28 registers are required.

(10) The abovementioned register-count requirement is satisfied by Fugaku, which has 32 registers.

(11) The ratio of registers to arithmetic operations is 28:16, which we assume to be equivalent to or greater than Condition (2) assuming the use of cache.

(12) Thus, the conditions of Items (3) and (4) are satisfied, and we expect the code to achieve high performance after tuning.

The performance effect of manual rescheduling is displayed graphically in Figure 5.87 and tabulated quantitatively in Figure 5.88. Subdividing the polynomial into mutually independent computational subtasks improves functional unit wait time by approximately 44%, from 532 μs to 296 μs. Additionally, the reduction in register spill/fill events decreases the number of load/store instructions, while overall runtime is improved by approximately 9%.

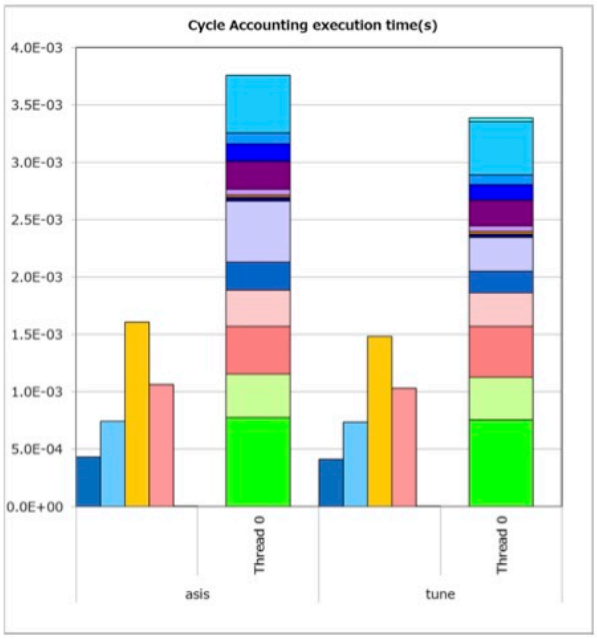


Figure 5.87: Performance charts showing the effect of manual rescheduling.

source code version	asis	tune
tuning contents	-	manually scheduled
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	3.75E−03	3.39E−03
GFLOPS/process	100.59	111.04
memory throughput (R+W) GB/s/process	0.005	0.006
float. point ops/thread	3.13E+07	3.15E+07
executed instructions/thread	5.89E+06	5.27E+06
load/store instructions/thread	2.15E+06	1.94E+06
L1D misses/thread	1.22E+05	1.21E+05
L2 misses/thread	1.40E+01	1.40E+01
L1 busy ratio	40.55%	41.87%
L2 busy ratio	28.51%	30.26%
memory busy ratio	0.00%	0.00%

Figure 5.88: Performance data showing the effect of manual rescheduling.

5.4.13 Eliminating SIMD reductions to decrease instruction count

The applying SIMD to an inner-product (vector-vector product) calculation increases the number of instructions executed due to SIMD handling of component-component reductions, as shown in Figure 5.89. In this figure, red-shaded regions indicate the contributions of redundant or unnecessary instructions to the instruction count growth. The effect of this instruction count increase is particularly significant on wide-SIMD machines such as Fugaku.

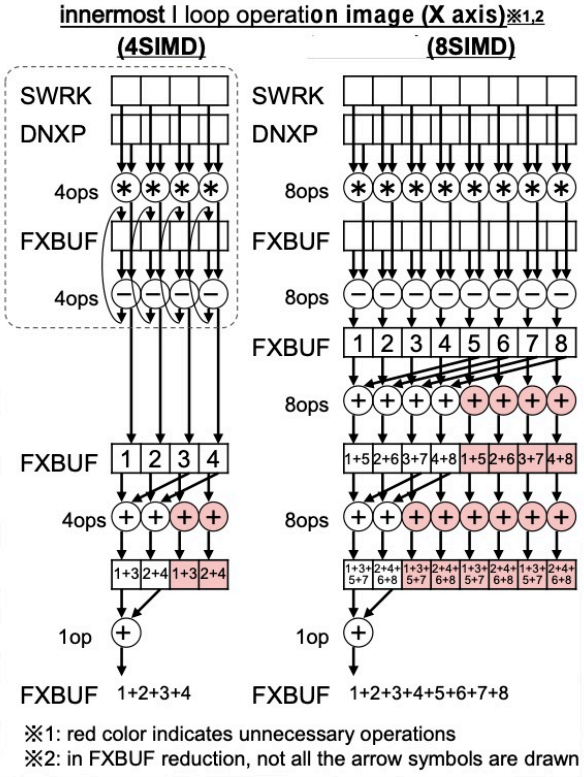


Figure 5.89: SIMD handling of component-component reductions.

Therefore, it is best to reduce SIMD handling of inner products as much as possible. More specifically, one

might consider computing the inner product of two long vectors via the method outlined in Figure 5.90, in which the long vectors are decomposed into smaller vectors of appropriate lengths, whose inner products are separately computed and later summed to yield a single scalar result. However, if we are attempting to minimize the use of reduction operations, we might proceed instead via the more efficient method outlined in Figure 5.91. Here, the long vectors are again subdivided into smaller vectors of appropriate lengths, but then accumulation operations are performed with the vector format retained, executing only a single reduction operation at the end of the calculation to yield the scalar result.

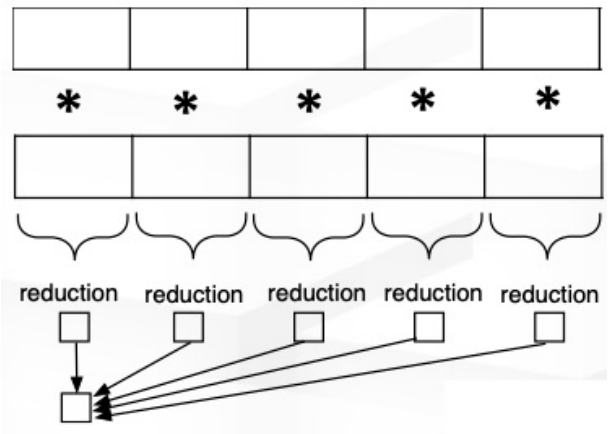


Figure 5.90: Schematic diagram illustrating one approach to inner-product calculation.

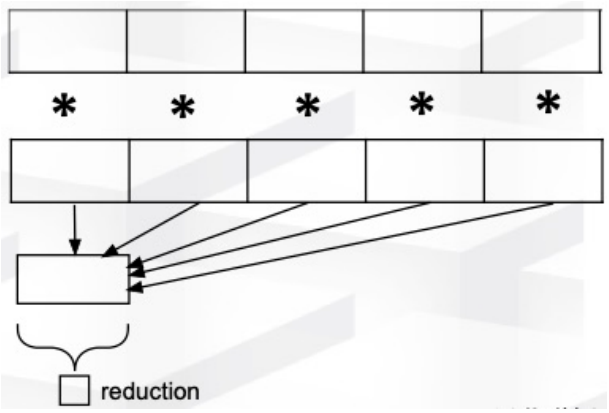


Figure 5.91: Schematic diagram illustrating an alternative approach to inner-product calculation.

The performance effect of eliminating SIMD reductions to decrease the instruction count is displayed graphically in Figure 5.92 and tabulated quantitatively in Figure 5.93. In these figures, it can be seen that numbers of floating-point operations and executed instructions are both significantly reduced, and that overall execution time decreases by approximately 22%. Additionally, memory busy time increases to a high value, and memory throughput grows to around 213 GB/s, which is close to the theoretical limits.

5.5 Tuning Technique 4: Making data access contiguous

The tuning technique discussed in this section is applicable to in-cache applications with low B/F ratios for which optimal tuning is possible (Figure 5.94).

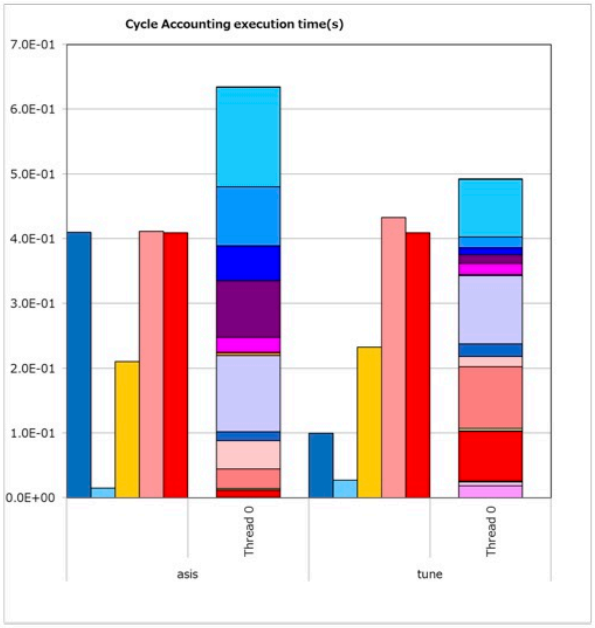


Figure 5.92: Performance charts showing the effect of eliminating SIMD reductions to decrease the instruction count.

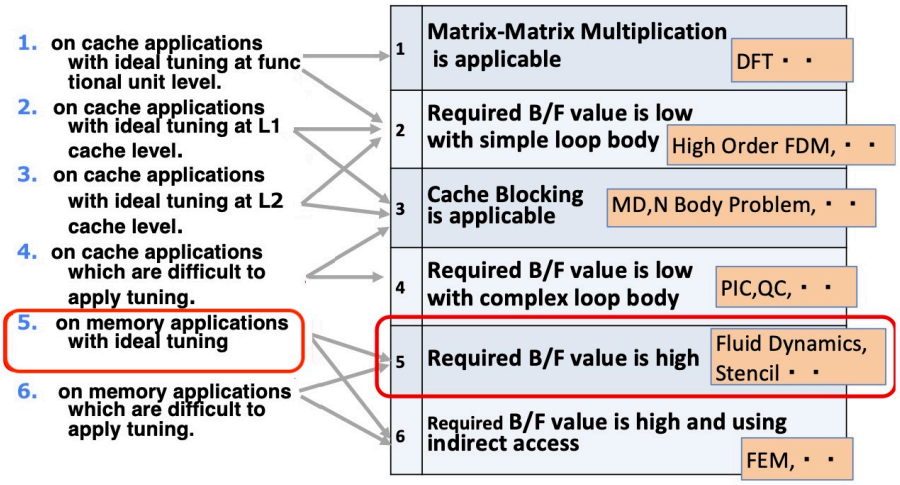


Figure 5.94: Application categories to which Tuning Technique 4 is applicable.

5.5.1 Making data access contiguous: Method 1

When the full range of a multidimensional FORTRAN array is accessed sequentially, the multidimensional array can be treated as a 1D array to facilitate efficient contiguous storage. In the example of Figure 5.95, we create a new subroutine to initialize the `rg` array, take the formal parameters to have the shape of a 1D array, and then construct a loop to initialize it.

source code version	asis	tune
tuning contents	–	avoid SIMD reduction
compier version	tcsds-1.2.27	tcsds-1.2.27
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	double	double
gathering thread #	0	0
elapsed time[s]	6.33E-01	4.92E-01
GFLOPS/process	163.76	137.90
memory throughput (R+W) GB/s/process	165.23	212.79
float. point ops/thread	8.70E+09	5.69E+09
executed instructions/thread	2.16E+09	8.83E+08
load/store instructions/thread	3.06E+08	2.78E+08
L1D misses/thread	4.79E+07	4.80E+07
L2 misses/thread	3.40E+07	3.39E+07
L1 busy ratio	33.43%	47.23%
L2 busy ratio	64.93%	87.97%
memory busy ratio	64.54%	83.12%
SFI ratio/Store-Fetch	0.0235	0.0169

Figure 5.93: Performance data showing the effect of eliminating SIMD reductions to decrease the instruction count.

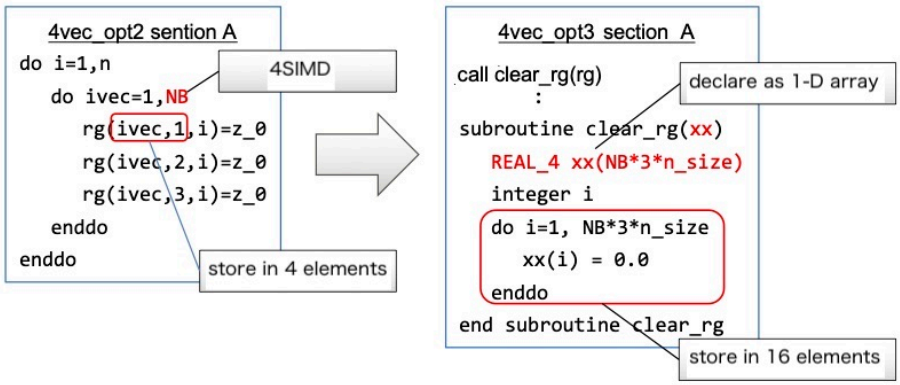


Figure 5.95: Making data accesses contiguous: Method 1 (`4vec_opt3`).

The performance effect of Method 1 for making data access contiguous is displayed graphically in Figure 5.96. Here, it can be seen that busy time increases, and that memory throughput rises to levels close to the theoretical limits.

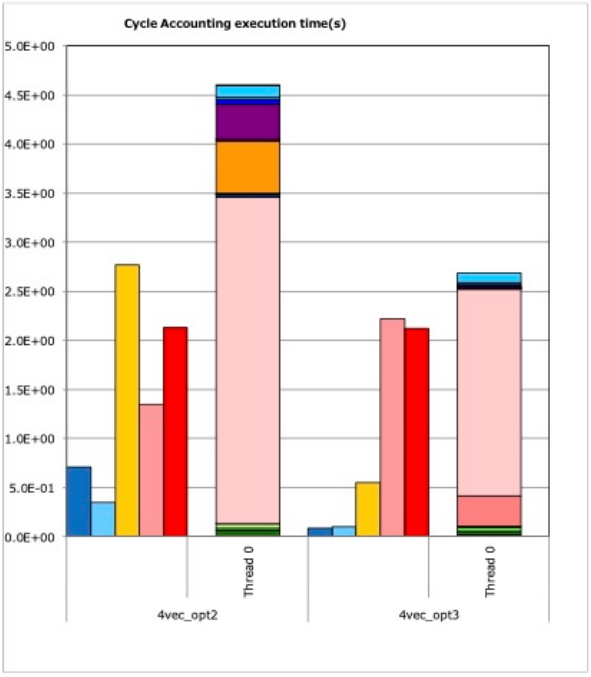


Figure 5.96: Performance charts showing the effect of Method 1 for making data access contiguous.

5.5.2 Making data access contiguous: Method 2

Filling zero values is frequently required in many applications, and in some applications its cost can be non-negligible. For such cases, system provided routines such as `memset` can provide efficient continuous zero value filling operations. Use `memset` in combination with `zfill` (`the DC ZVA` instruction), thereby increasing efficiency. This is shown by the program presented in Figure 5.97.

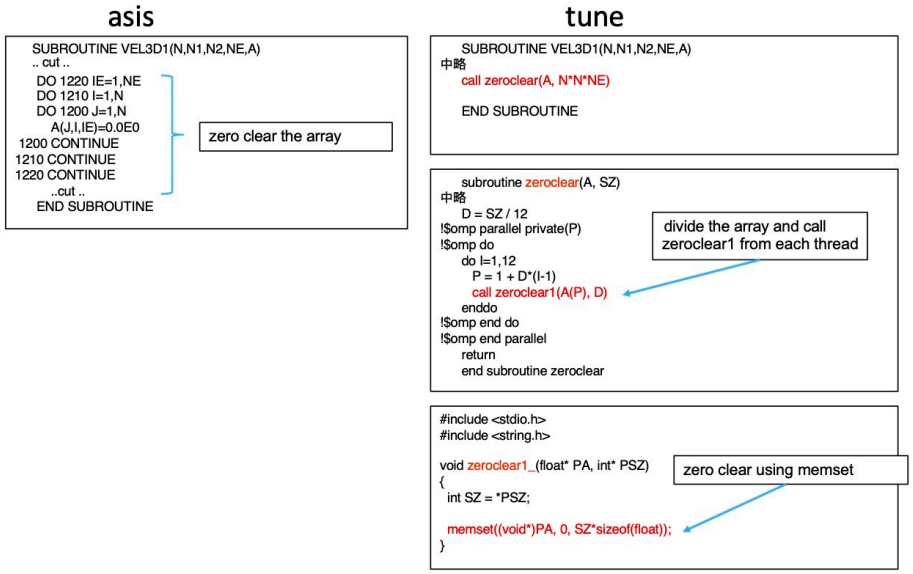


Figure 5.97: Making data access contiguous: Method 2

The performance effect of Method 2 for making data access contiguous is displayed graphically in Figure 5.98 and tabulated quantitatively in Figure 5.99. These figures show that instruction count and the number of

cache misses both decrease significantly, while the L2-cache busy ratio rises to the high value of 82%, thus indicating performance close to the theoretical limits. Linking fast `memset` can be accomplished by providing `-Koptlib_string` option to the compiler/linker.

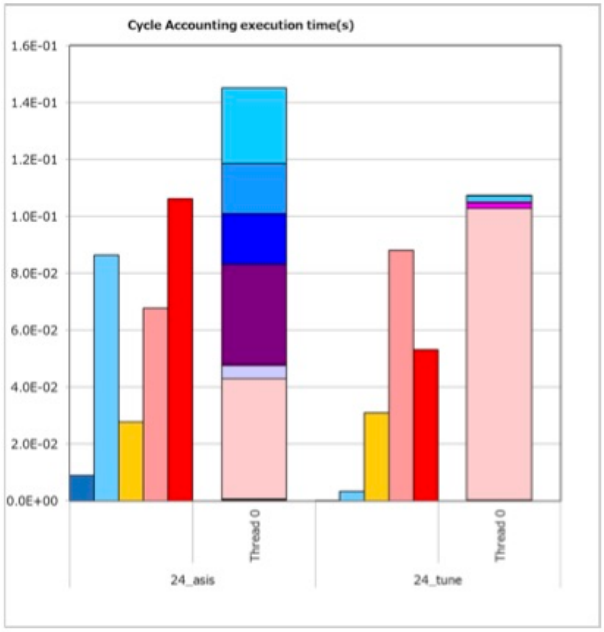


Figure 5.98: Performance charts showing the effect of Method 2 for making data access contiguous.

source code version	asis	tune
tuning contents	-	consecutive data access
compier version	tcsds-1.2.27b	tcsds-1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	1.45E-01	1.07E-01
GFLOPS/process	0.00	0.00
memory throughput (R+W) GB/s/process	187.30	126.79
float. point ops/thread	0.00E+00	0.00E+00
executed instructions/thread	4.60E+08	1.79E+07
load/store instructions/thread	3.55E+07	4.50E+06
L1D misses/thread	4.43E+06	2.87E+03
L2 misses/thread	4.43E+06	1.22E+03
L1 busy ratio	19.14%	28.86%
L2 busy ratio	46.60%	82.29%
memory busy ratio	73.16%	49.53%
DCZVA instructions	3.84E+02	5.31E+07

Figure 5.99: Performance data showing the effect of Method 2 for making data access contiguous.

5.5.3 Making data access contiguous: Method 3

The third example for making data access contiguous is to replace the `gather load` and `scatter store` instructions with contiguous memory accesses. This is shown by the program seen in Figure 5.100. Whereas the original (unmodified) program accesses memory in blocks of size three, the tuning process rewrites the code to use contiguous access instead.

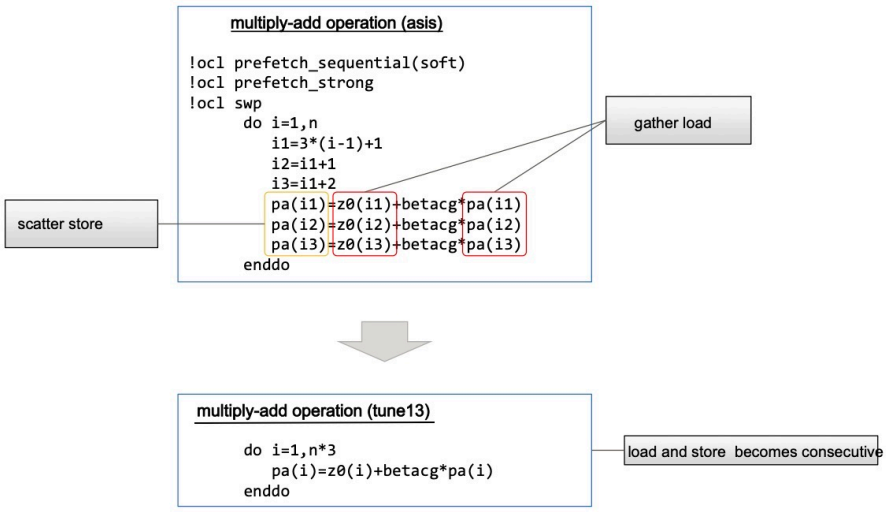


Figure 5.100: Making data access contiguous: Method 3 (`tune13`).

The performance effect of Method 3 for making data access contiguous is displayed graphically in Figure 5.101 and tabulated quantitatively in Figure 5.102. Looking at the breakdown of the load/store instruction count, we see that `gather load` and `scatter store` instructions have been replaced with contiguous loads and stores. Because `scatter store` instructions are no longer needed, prefetch instructions are also no longer needed. Instead, hardware prefetching is used. Eliminating the computation of `gather load` and `scatter store` addresses further reduces the instruction count. The final result is that we achieve performance at levels very near the theoretical upper limit for memory bandwidth.

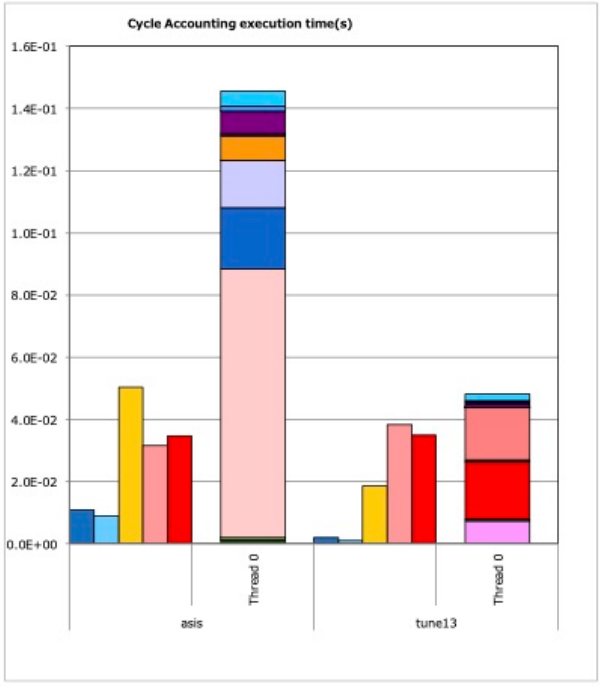


Figure 5.101: Performance charts showing the effect of Method 3 for making data access contiguous.

source code version	asis	tune13
tuning contents	–	consecutive data access
compier version	tcsu1.2.27b	tcsu1.2.27b
frequency(GHz)	2.0	2.0
kernel coded	yes/no	yes/no
float. point precision	single	single
gathering thread #	0	0
elapsed time[s]	1.45E−01	4.69E−02
GFLOPS/process	10.52	32.66
memory throughput (R+W) GB/s/process	61.05	191.70
float. point ops/thread	1.28E+08	1.27E+08
executed instructions/thread	6.23E+07	2.03E+07
load/store instructions/thread	1.23E+07	1.22E+07
L1D misses/thread	2.02E+06	2.03E+06
L2 misses/thread	2.01E+06	2.05E+06
L1 busy ratio	34.79%	39.12%
L2 busy ratio	21.80%	81.91%
memory busy ratio	23.85%	74.88%
consecutive load instructions/thread	2.12E+02	7.97E+06
gather load instructions/thread	7.97E+06	0.00E+00
consecutive store instructions/thread	2.08E+02	3.98E+06
scatter store instructions/thread	3.98E+06	0.00E+00

Figure 5.102: Performance data showing the effect of Method 3 for making data access contiguous.

5.6 Tuning Technique 5: Avoiding excessive SFI

In this section, we present a tuning technique that addresses the performance-degrading phenomenon known as *excessive store/fetch interlock* , aka excessive SFI, a problem that can arise for applications in any of the categories we have discussed.

5.6.1 Avoiding excessive SFI

Let's consider an array `A` whose layout in memory corresponds to a nested loop structure, and suppose the innermost loop has length six. Because six is less than the SIMD width of eight, applying SIMD to the inner loop involves the use of compiler directives to *mask* the two leftover SIMD slots. Now, suppose an accumulation operation is performed to add a second array to array `A`. Because this is an accumulation, `A` appears both as the array from which values are loaded on the right-hand side and the array to which values are stored on the left-hand side. In cases like this, SIMD is used for the innermost loop, and because the outer loop (whose index we call i) typically does not involve recurrence, instruction scheduling can be used to achieve overlapping execution.

Inspecting the pattern of memory accesses generated in this case reveals that the storage operation for iteration i overlaps with the load operation for iteration $i + 1$ and the masked SIMD slots. This situation is shown in Figure 5.103.

Logically, this overlap should not prevent the two operations from proceeding simultaneously. However, the hardware is designed to ensure that loads do not precede the stores if the masked portion between the overlap region containing the stores, which is called store fetch interlock. This SFI is a hardware feature to assure the safe loads and stores. There can be an occasion where SFI is activated more than necessary, which is called excessive SFI. If excessive SFI occurs, load and store instructions are not scheduled optimally, and affect pipeline efficiency, thus causing performance degradation.

Excessive SFI can be avoided by the use of array padding, as demonstrated by the code of Figure 5.104. In this example, the second and third loops are candidates for possible SFI activation, so we pad the `tmp` and `result2` arrays.

The performance effect of excessive SFI avoidance is displayed graphically in Figure 5.105 and tabulated quantitatively in Figure 5.106. These figures show that the frequency of SFI occurrence is greatly reduced, as are the wait times for L1D cache access and functional unit operations, thereby cutting overall execution time by 87%. The memory busy ratio increases significantly, and memory throughput rises to 213 GB/s, thus approaching its theoretical upper bound.

Compilers have been enhanced to apply SWPL and scheduling operations in advance so that the $(i + 1)$th load instruction is issued after the ith store instruction, the occasions of excessive SFI situations have been

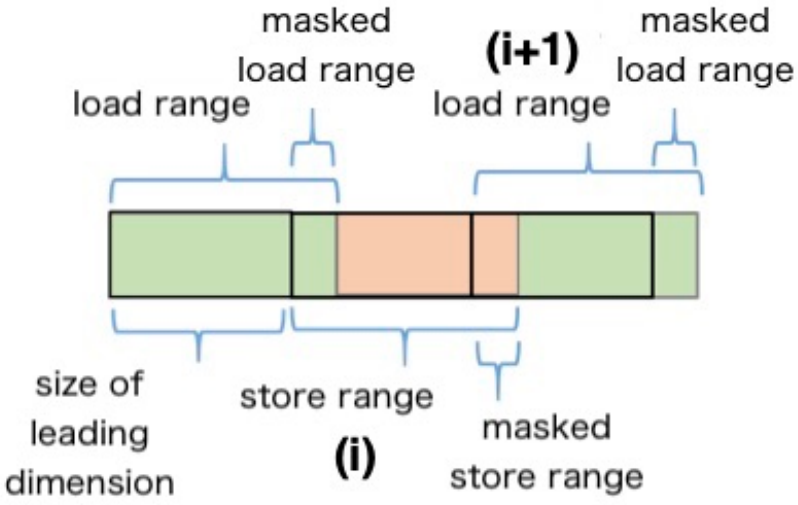


Figure 5.103: Overlap between storage for iteration i and loading for iteration $i+1$.

reduced. Nonetheless, in case the the emergence of excessive SFI is observed, the tuning technique described above maybe useful. SFI values are included in CPU performance analysis reports. If the value is high, it should trigger suspicions that excessive SFI is occurring.

5.6.2 Avoiding excessive SFI: Consequences of aggregating `gather load` instructions

When arrays are accessed indirectly using index list, compilers will generate `gather load` instructions. For these `gather load` instructions, two adjacent elements belonging to the same 128-byte block can be combined and handled in a single flow. In the address-pattern example shown in Figure 5.107, the items outlined in blue are aggregated for loading together as two elements.

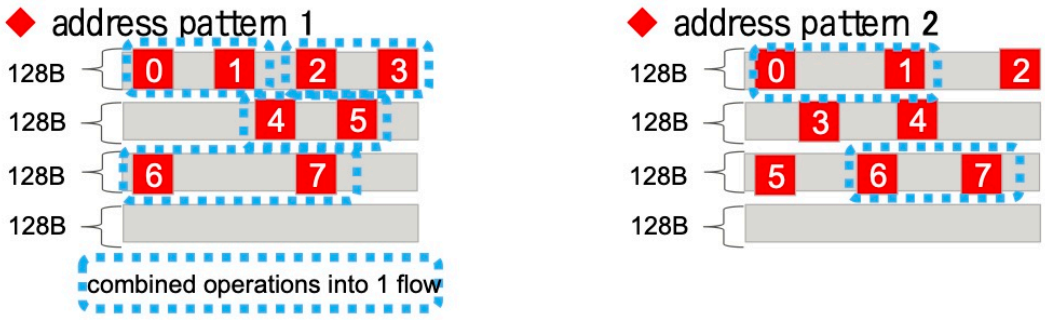


Figure 5.107: Aggregating `gather load` instructions.

In array accumulation operations, a list-accessed array can appear on both the right- and left-hand sides of arithmetic operations, as shown in the code outlined in Figure 5.108. In such cases, the `gather load` instructions described above are used and can be aggregated.

Arrays `list1` through `list5`, which store list indices used to access array `F`, are guaranteed not to contain two or more copies of any one index value. Additionally, we assume that the addresses used in the loop to access `F` elements are known never to coincide. The absence of identical indices does not preclude the presence of nearby indices, so insertions into the `F` array in Figure 5.108 can generate data storage on identical cache lines. If this happens and aggregation procedures are in effect, excessive SFI will result, thus preventing loads from preceding stores and potentially degrading performance.

```
void MatVec_product(MatVec *self, double vector[], double result_OUT[]){
static double tmp[N_THREADS][MAX_DIAG_BLOCKS][6];    // padding 6→8
static double vector2[MAX_DIAG_BLOCKS][6];
static double result2[MAX_DIAG_BLOCKS][6];                    //padding 6→8
static double vec_c[6]
for (jBlock = 0; jBlock < iBlock; jBlock++) {
…
#pragma loop simd_redundant_vl 6
      for ( iv = 0; iv < 6; iv++ ){
         double v_j = vector2[jBlock][iv];
         double m_i0_j = OffDiagComponents[instanceId][offset + jBlock][0][iv];
         double m_i1_j = OffDiagComponents[instanceId][offset + jBlock][1][iv];
         ...
         vec_c[0] += m_i0_j * v_j;
         vec_c[1] += m_i1_j * v_j;
         ...
      }
#pragma loop simd_redundant_vl6
      for ( iv = 0; iv < 6; iv++ ){
         double m_i0_j = OffDiagComponents[instanceId][offset + jBlock][0][iv];
         double m_i1_j = OffDiagComponents[instanceId][offset + jBlock][1][iv];
         ...
         tmp[threadId][jBlock][iv] += m_i0_j * v_i0 + m_i1_j * v_i1 + m_i2_j * v_i2
            + m_i3_j * v_i3 + m_i4_j * v_i4 + m_i5_j * v_i5;
       }
} /* jblock */
…
for (iv = 0; iv <6; iv++) { result2[iBlock][iv] += tmp[iThread][iBlock][iv]; }
}
```

Figure 5.104: Program demonstrating padding used to prevent excessive SFI.

Whenever possible, present-day compilers use SWPL or other scheduling methods to ensure that the `gather load` instruction on the second line here is generated before the `scatter store` instruction on the first line, so this situation rarely gives rise to excessive SFI. However, since scheduling may not be possible due to a shortage of registers or other factors, it can be useful, in such cases, to tune the codes so that loads explicitly precede stores, as shown in Figure 5.109.

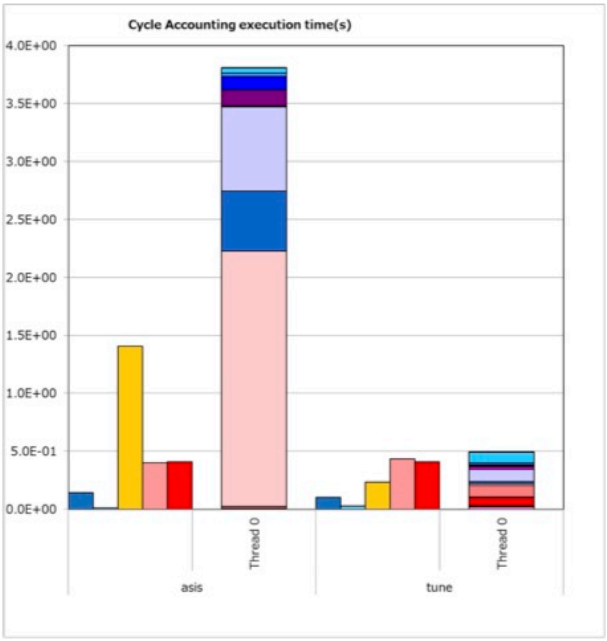


Figure 5.105: Performance charts showing the effect of excessive SFI prevention.

source code version	asis	Tune
tuning contents	-	array padding
compier version	tcsds-1.2.27	tcsds-1.2.27
frequency(GHz)	2.0	2.0
kernel coded	yes	yes
float. point precision	double	double
gathering thread #	0	0
elapsed time[s]	3.81E+00	4.92E-01
GFLOPS/process	17.82	137.90
memory throughput (R+W) GB/s/process	27.38	212.79
float. point ops/thread	5.69E+09	5.69E+09
executed instructions/thread	1.28E+09	8.83E+08
load/store instructions/thread	6.62E+08	2.78E+08
L1D misses/thread	4.62E+07	4.80E+07
L2 misses/thread	3.40E+07	3.39E+07
L1 busy ratio	37.25%	47.23%
L2 busy ratio	10.56%	87.97%
memory busy ratio	10.70%	83.12%
SFI ratio/Store-Fetch	0.5583	0.0169

Figure 5.106: Performance data showing the effect of excessive SFI prevention.

```
do i=1,n
 do j=1,m

 . . .
  F(j,list1(i)) = F(j,list1(i))+WORK1(i)
  F(j,list2(i)) = F(j,list2(i))+WORK2(i)
  F(j,list3(i)) = F(j,list3(i))+WORK3(i)
  F(j,list4(i)) = F(j,list4(i))+WORK4(i)
  F(j,list5(i)) = F(j,list5(i))+WORK5(i)
 . . .
 end do
end do
```

Figure 5.108: An example of a program in which aggregating `gather load` instructions can cause excessive SFI.

```
do i=1,n
 do j=1,m

 . . .
  t1 = F(j,list1(i))
  t2 = F(j,list2(i))
  t3 = F(j,list1(i))
  t4 = F(j,list1(i))
  t5 = F(j,list1(i))
  F(j,list1(i))= t1 + WORK1(i)
  F(j,list2(i))= t2 + WORK2(i)
  F(j,list3(i))= t3 + WORK3(i)
  F(j,list4(i))= t4 + WORK4(i)
  F(j,list5(i))= t5 + WORK5(i)
 . . . .
 end do
end do
```

Figure 5.109: Program demonstrating tuning via aggregation of `gather load` instructions to avoid excessive SFI.