

# Analysis and Optimization of Yee\_Bench Using Hardware Performance Counters

Ulf Andersson<sup>a</sup>, Philip Mucci<sup>a,b</sup>

<sup>a</sup>PDC, KTH, SE-100 44 Stockholm, Sweden, ulfa@nada.kth.se

<sup>b</sup>Innovative Computing Laboratory, University of Tennessee, mucchi@cs.utk.edu

In this paper, we report on our analysis and optimization of a serial Fortran 90 benchmark called `Yee_bench`. This benchmark has been run on a variety of architectures and its performance is reasonably well understood. However, on AMD Opteron based machines, we found unexpected dips in the delivered MFLOPS of the code for a seemingly random set of problem sizes. Through the use of the Opteron's on-chip hardware performance counters and `PapiEx`, a PAPI based tool, we discovered that these drops were directly related to high L1 cache miss rates. The high miss rates could be attributed to the fact that in the two compute kernels of the application there are references to three equal-sized dynamically allocated arrays which compete for the same set in the Opteron's 2-way set-associative cache. We validated this conclusion by accurately predicting those problem sizes that exhibit this problem. Furthermore, we were able to alleviate these performance anomalies using variable intra-array padding to effectively accomplish inter-array padding. We conclude with some comments on the general applicability of this method as well how one might improve the implementation of the Fortran 90 `ALLOCATE` intrinsic to handle this and other cases of set-associativity conflicts.

## 1. Introduction

In this paper, we report on the analysis and optimization of `Yee_bench` [1,2] using a Performance Application Programming Interface (PAPI) [7] based tool called `PapiEx` (PAPI Execute) [8]. We show how we tracked down a number of performance dips when `Yee_bench` was run on an AMD Opteron and the problem size was varied.

`Yee_bench` is a benchmark developed at the Center for Parallel Computers (PDC) in Stockholm, Sweden. It implements the core of the Finite-Difference Time-Domain (FDTD) method [5,6,9] for the Maxwell equations, commonly used in computational electromagnetics. See Appendix A for a listing of the core of `Yee_bench`. The `Yee_bench` code is strongly bound by memory bandwidth and its results show a strong correspondence [1] with the numbers produced from the `STREAM2` microbenchmark [4]. `Yee_bench` is widely used at PDC as part of the architectural evaluation process when purchasing new hardware as many of our applications are bound by memory bandwidth.

To achieve the best possible speed-up and scale-up for the parallel version of `Yee_bench`, it is crucial to understand the performance of the serial code and how it depends on the problem size. If this behavior can be understood, we will know when to use padding to avoid unsuitable local problem sizes in the parallel code. Many FDTD applications strives to use as much memory as possible, hence scale-up is a relevant measure of parallel performance. Thus it follows that behavior for large problem sizes are more important than performance for small problem sizes.

These dips in performance occurred with no apparent regularity and thus we looked to hardware performance monitors (through the use of PAPI and `PapiEx`) to provide us with additional insight. `PapiEx` is a portable and easy-to-use command line tool to monitor the performance counters for

any executable. No source code instrumentation is needed although a simple instrumentation API is provided. Through the information gathered with `PapiEx`, we were able to characterize and subsequently alleviate these performance anomalies in `Yee_bench`.

We also present the performance of `Yee_bench` on the Intel Itanium-2 and the Intel Xeon EM64T for completeness.

## 2. AMD Opteron

### 2.1. Technical data

For our AMD runs we used one CPU of a four-way Opteron Processor 846 running at 2.0 GHz. The memory subsystem consists of a two-way set-associative 64 kbytes L1 cache, a sixteen-way set-associative 1 Mbytes L2 cache, and 8 Gbytes of DDR 333 with 2 Gbytes per CPU. The L1 cache line length was 64 bytes.

The compiler used was version 5.2-4 of `pgf90`. The following options were used: `-fast -fastsse -tp=amd`. Similar behavior was observed with the `pathf90` compiler on other Opteron systems. The OS used was SUSE 9.1 with Linux kernel 2.6.8.

### 2.2. Results

The upper part of Figure 1 displays the 64-bit precision performance of the original (no padding) version of `Yee_bench` [1]. The computational domain used is a cube and thus  $N_x = N_y = N_z \equiv N$ .

The results for the original code in Figure 1 are representative of all Opteron results achieved for `Yee_bench`. Based on our previous experience on other architectures, we expect poor performance whenever  $N$  or  $N + 1$  is a power-of-two if no padding is used [1]. On some architectures we get poor performance whenever  $N$  or  $N + 1$  contains at least four factors of 2 due to low cache hit rate (see [1]). This phenomena is well understood and easy to avoid using padding. However, in Figure 1 we have poor performance for many more problem sizes than previously encountered.

### 2.3. L1 Cache hit rate

We initially suspected intra-array way conflicts in the L1 cache. So we ran a different version of `Yee_bench`, one that contained intra-array padding. Each of the three electric field arrays were padded identically, as were the three magnetic field arrays. We also tried the stock version of the code, except that we introduced compiler directives to do the padding. Neither had any appreciable affect on performance, which ruled out intra-array conflicts. However, such tremendous drops in performance could only be caused by L1 misses. We therefore decided to measure the L1 cache hit rate for all problem sizes. The result is displayed in the lower part of Figure 1. The L1 cache hit rate was computed from `PapiEx` output as:

$$\frac{\text{PAPI\_L1\_DCH}}{\text{PAPI\_L1\_DCH} + \text{PAPI\_L1\_DCM}} \quad (1)$$

There is clearly a correspondence between performance (GFLOPS) and L1 cache hit rate. The question now becomes: Why do we get so low L1 cache hit rate for certain problem sizes?

### 2.4. L1 cache hit rate analysis

We will first determine what is the best possible L1 cache hit rate for the loops (listed in Appendix A) in the kernel of `Yee_bench`. Each iteration of these triple nested loops contains three stores and ten loads. Four of the ten loads have already been used in earlier iterations, while six of the ten loads are new. Actually, there appear to be two more ‘‘loads’’, but these values were used in

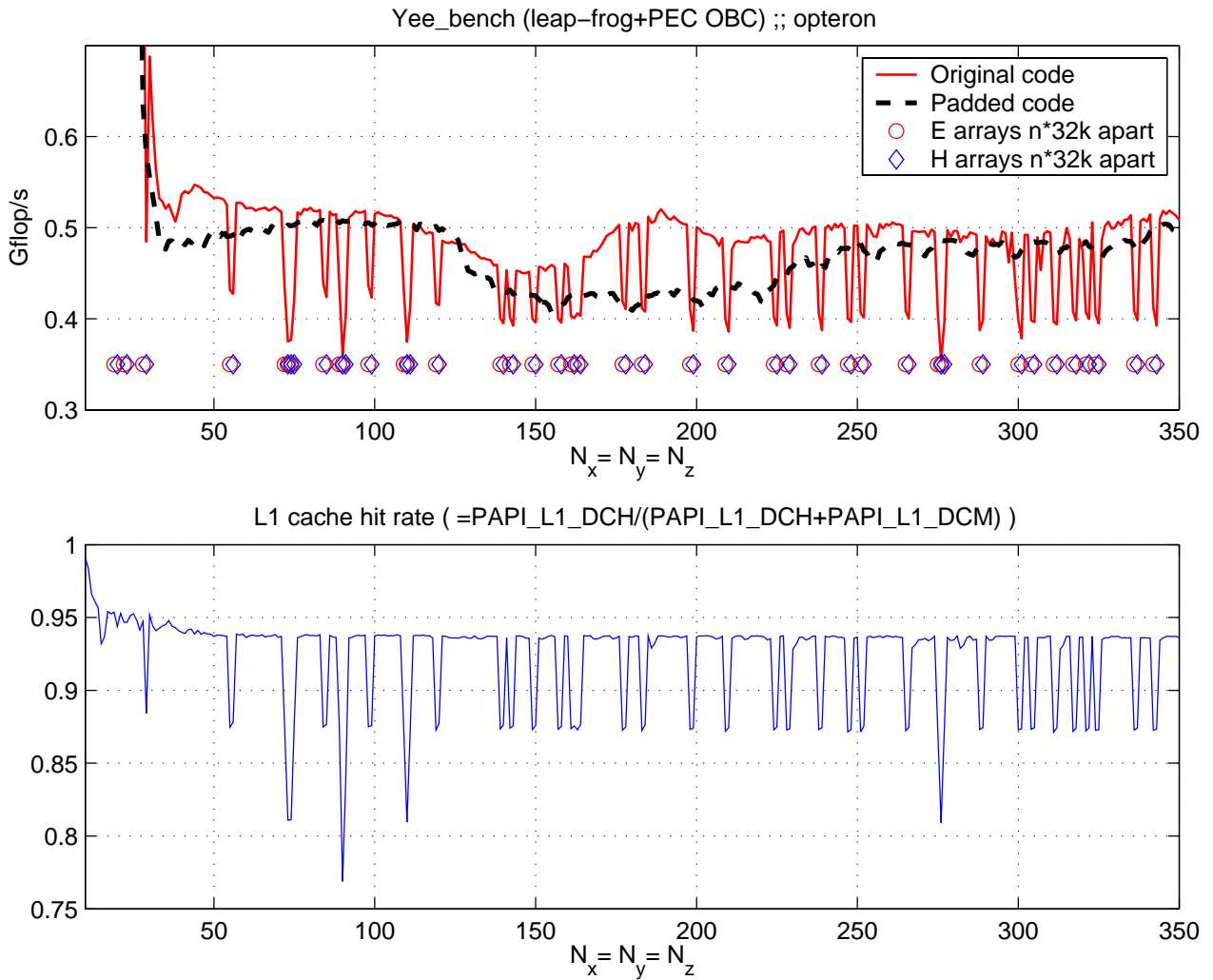


Figure 1. Yee\_bench performance and L1 cache hit rate on the AMD Opteron.

the preceding iteration and can therefore be expected to reside in registers. This conjecture has been verified on several computers using hardware performance counters.

Each L1 cache line contains eight 64-bit precision floating-point values. This means that one of eight loads is a compulsory cache miss for each of the six new values. The four loads of already used values will always, in the ideal case, already be in cache. We expect every store to be a cache hit. This gives us a load hit rate of:

$$\frac{7}{8} \frac{6}{10} + \frac{4}{10} = \frac{37}{40} \approx 92.5\%, \quad (2)$$

and a total L1 cache hit rate of:

$$\frac{10}{13} \frac{37}{40} + \frac{3}{13} \approx 94.2\%. \quad (3)$$

The measured L1 cache hit rate for the good cases in Figure 1 is around 93.7%, which is pretty close to our estimate in (3), which is an upper limit. It is not reasonable to expect the four loads of already used values to always be cache hits due to capacity limitations. Two of these values was used

in the preceding iteration for the middle loop and two values were used in the preceding iteration for the outer loop. Hence they may have been evicted from cache.

If none of the ten loads give a hit when we load the first element of a cache line we get a load hit rate of 7/8 and a total hit rate of:

$$\frac{10}{13} \cdot \frac{7}{8} + \frac{3}{13} \approx 90.4\% \quad (4)$$

The low cache hit rate values in Figure 1 are below the estimate in (4) which indicates that we have cache line contention. In the following section we will explore whether it is possible to predict for which values of  $N$  this will happen.

## 2.5. Allocation analysis

By using the utility function `LOC()`, which returns the address of a data item, we established that the electromagnetic arrays are always allocated with a distance that is a multiple of the pagesize (4 kbytes) when the arrays are larger than 128 kbytes. Considering that the L1 cache is two-way set-associative, we notice that if the three cache lines containing  $E_x(i, j, k)$ ,  $E_y(i, j, k)$  and  $E_z(i, j, k)$  belong to the same set, we have contention for the same cache line.

The L1 cache is 64 kbytes. If we divide this by two due to the set-associativity, we get 32 kbytes or eight pages. Hence we postulate that we will get poor performance if the distance between the different arrays is a multiple of 32 kbytes. In Figure 1, we have indicated for which problem sizes we expect this to happen. To be able to do this prediction, it is important to note that there is a bit of overhead on each allocation. We have measured this overhead to be 80 bytes.

## 2.6. Padding

The original version of `Yee_bench` allowed for internal padding of the electric and magnetic field arrays. However, it used the same padding on all three electric field arrays and similarly for the three magnetic field arrays. This padding procedure was used in order to avoid having leading dimension that were powers-of-two. Due to cache-line contention, it was necessary to find a way to make sure that the iterations over the three dimensional arrays didn't start at the same place. However, the `ALLOCATE` statement in both the Portland Group compiler and the Pathscale compiler always returned data on a page boundary. In order to reduce the amount of changes to the code, we decided to use the existing padding infrastructure but pad each array differently. By doing so, we could guarantee that each iteration, except the first iteration, computed at different offsets into each array. With these changes, contention should only occur during the first iteration of the inner loop (see Appendix A) and the padding would handle subsequent iterations.

To achieve the individual padding we used this allocation scheme:

```
Hx(1:nx +padHx(1), 1:ny +padHx(2), 1:nz +padHx(3))
Hy(1:nx +padHy(1), 1:ny +padHy(2), 1:nz +padHy(3))
Hz(1:nx +padHz(1), 1:ny +padHz(2), 1:nz +padHz(3))
Ex(1:nx+1+padEx(1), 1:ny+1+padEx(2), 1:nz+1+padEx(3))
Ey(1:nx+1+padEy(1), 1:ny+1+padEy(2), 1:nz+1+padEy(3))
Ez(1:nx+1+padEz(1), 1:ny+1+padEz(2), 1:nz+1+padEz(3))
```

Figure 1 displays the results for  $\text{padEx}=\text{padHx}=(/1, 0, 0/)$ ,  $\text{padEy}=\text{padHy}=(/2, 0, 0/)$ , and  $\text{padEz}=\text{padHz}=(/3, 0, 0/)$ . We see that all the large dips have disappeared. However, for  $150 < N < 250$  we see a loss of performance compared to the good cases of the original code. For the most important problem sizes, the large ones, we see a considerable improvement in performance

for the padded code version. Since we are able to predict when padding is needed we can choose to use it only when it is needed.

Here we have effectively achieved inter-array padding by doing intra-array padding. While complex, this technique is reasonably portable and could be implemented easily by the compiler during its loop optimization phase.

### 2.7. Building a Better Fortran 90 ALLOCATE

At the time of this work, neither the Portland Group compiler nor the Pathscale compiler provided any mechanisms to do inter-array padding or otherwise change the behavior of the allocator. In theory, a compiler could be modified to provide hints to the allocator based on possible conflicts detected during the loop nest optimization phase. However, this would require significant complexity since the allocator could easily be in another module, requiring the modifications to be deferred until a whole program optimization phase (or global intraprocedural analysis phase). Additionally, this could cause significant overhead as a program could contain hundreds if not thousands of allocates and loop nests, each of which would have to be dealt with. Instead of making significant modifications to the compiler, it is a better choice to simply waste some memory and return locations offset by one or more cache lines. Algorithm 2.1 implements such a scheme by maintaining a static local variable that contains the number of lines in one page. For multithreaded programs, we need not be concerned with atomically updating the “pad” variable, as this would result in only the occasional allocation with the same padding. This allocator need not be called for smaller allocations, say, under one page.

#### Algorithm 2.1: ALLOCATE(*bytes*)

```

static pad ← 0
local total_bytes, num_pages_padded, leftover, mem, address

total_bytes ← bytes + pad * L1LINESIZE
num_pages_padded ← total_bytes / PAGESIZE
leftover ← total_bytes % PAGESIZE
if (leftover > 0)
    num_pages_padded ← num_pages_padded + 1
mem ← ALLOCATE_PAGES(num_pages_padded)
address ← mem + pad * L1LINESIZE
if (pad + 1 ≥ L1LINESPERPAGE)
    pad ← 0
else
    pad ← pad + 1

return (address)

```

### 2.8. Comments on results from other AMD Opteron systems

The performance loss when we get cache line contention is about 20% for the original code in Figure 1. On other Opteron systems we have seen as much as 50% performance loss, thus making it a very severe issue.

### 3. Intel Itanium-2

The 64-bit precision results for `Yee_bench` on the Intel 900 MHz Itanium-2 are shown in Figure 2. There is a clear correspondence between the performance and the L3 cache hit rate. The L2 cache size was 256 kbytes and the L3 cache size was 1.5 Mbytes. (The L1 cache is not used by floating-point data.)

The L3 cache is referenced whenever we have an L2 cache miss. A majority of the L2 cache misses are compulsory cache misses and therefore become L3 cache misses unless the problem size is small. As mentioned in Section 2.4, we have ten loads per iteration. Six of these loads are compulsory cache misses, two of these loads are highly likely to hit L2 cache, since they were used for the previous iteration value of the middle iteration. Two of the loads refers to values that were used in the previous iteration of the outer loop. These may hit L2 cache. If they miss L2, they might hit L3. For large problem sizes, they will miss both, and we will get an L3 cache hit rate that is almost zero. For  $N = 100$  they will almost always miss L2 but hit L3 and we will get an L3 hit rate of 25% (two hits per six compulsory misses). The data traversed during one iteration of the outer loop is  $64N^2$  bytes. For  $N = 100$  this becomes 0.61 Mbytes, which is well within the L3 cache size.

The reason for the increase in performance when going from  $N = 45$  to  $N = 100$  is the loop restart penalty for the inner loop [2]. Examining the assembly code, we find that the *prolog* (and the *epilog*) consists of four iterations (see Chapter 2 in [3] for an explanation of the terms *prolog* and *epilog*). This is independent of  $N$  and is thus more expensive for smaller values of  $N$ .

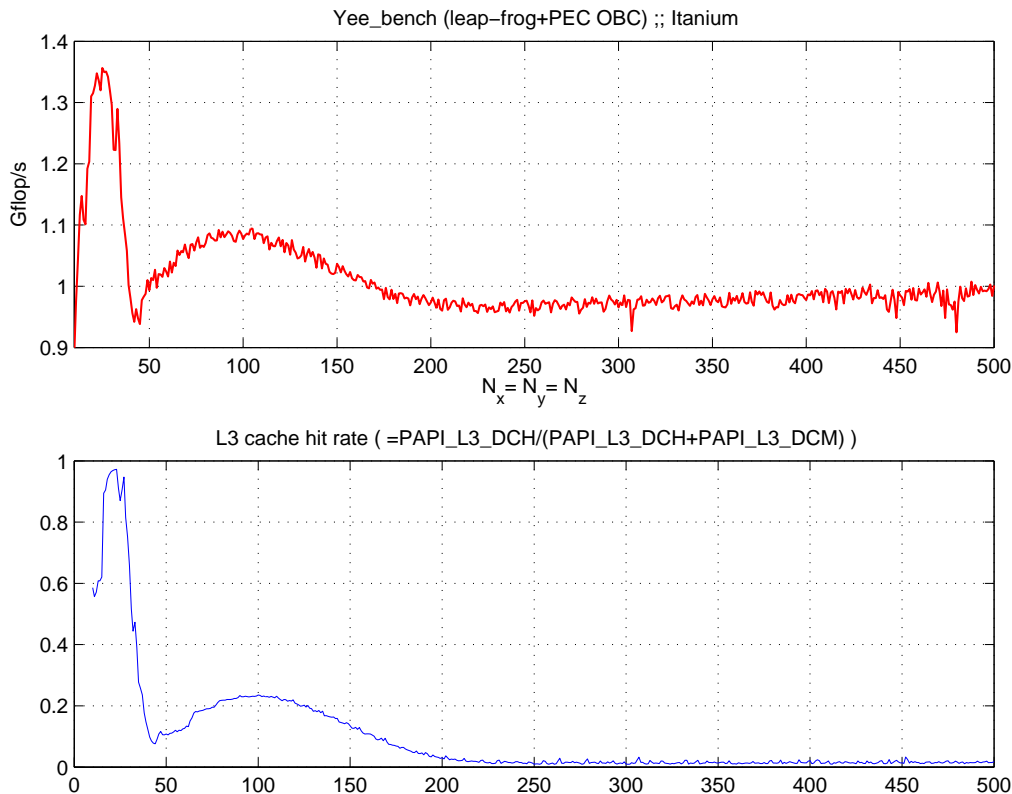


Figure 2. `Yee_bench` performance on the Intel Itanium-2.

#### 4. Intel Xeon EM64

Figure 3 displays the 64-bit precision performance of `Yee_bench` on the Intel 3.4 MHz Xeon EM64. Results are compared for the original and the padded code. The padding used was `padEx=padHx=( / 0 , 0 , 0 / )`, `padEy=padHy=( / 1 , 0 , 0 / )`, and `padEz=padHz=( / 2 , 0 , 0 / )`. Again we see that padding smooths the performance curve, but results in a slight performance loss for medium sized ( $30 < N < 100$ ) problems. Problem sizes smaller than  $N = 28$  fits into the one Mbyte L2 cache for the unpadded code.

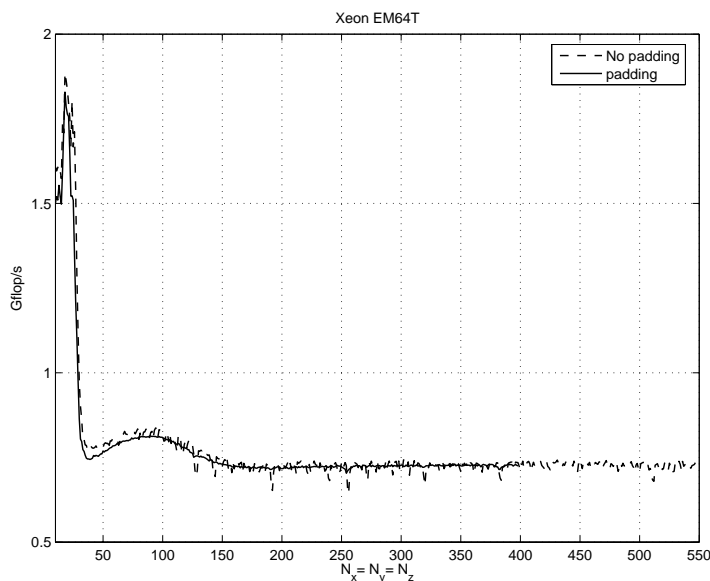


Figure 3. `Yee_bench` performance on the Intel Xeon EM64.

#### 5. Variable Intra-Array Padding as a Technique for Padding Dynamic Data Structures

Stepping back, it is clear that what makes this problem interesting is that these arrays are allocated dynamically and thus we have no direct control over the starting address. As more and more HPC applications move to Fortran 90, traditional approaches like common block padding and power-of-two avoidance become obsolete. There is nothing special about `Yee_bench` that makes it amenable to the solution we have put in place here, except for compile time constants for variable amounts of padding. If a compiler were instead to do this work at allocate time, it would have to be able to discover array references with possible conflicts and then rewrite their corresponding allocate statements. This is only possible with significant work in an intra-procedural analysis phase, which some compilers lack entirely. Replacing `ALLOCATE`, as discussed in Section 2.7, is an option, but that requires knowledge of the compiler's run-time system and the exact arguments and linkage of the call to the `ALLOCATE` intrinsic. Instead of the above approaches, consider that the compiler could generate code for array declarations such that every array in the application was padded by some pseudo-random amount of at least a cache line. This would include both static and dynamic allocations. This could be taken a step further and pad the starting address of each array by a similar amount. This approach drastically reduces the chances of a way conflict. Whether done by compiler or by hand, we believe that variable padding scheme method is generalizable to other codes.

## 6. Comments on the Tuning Process

One of the important lessons to be learned here is that we were able to gather hardware performance data with PapiEx that directly correlated with the codes performance. As this was a benchmark, it contained internal timers to compute a theoretical GFLOPS number. The code could have just as easily reported its performance in terms of timesteps/day or cell-domains/second. What was important was that the code's performance metric and the metric chosen for analysis (L1 miss rate) were both rates and thus independent of problem size.

Another important point is that the fact that no source code instrumentation is needed to use PapiEx means that an investigation of strange performance can be started immediately and that the executable was not perturbed in any way through instrumentation.

### A. The core of Yee\_bench

The core of Yee\_bench is two triple-nested loops, one for updating the magnetic field components and one for updating the electric field components. Close to 100% of the time is spent in these two triple-nested loops if the surrounding time-stepping loop contains enough time steps to dominate the initialization time. The code for the update of the magnetic field components is:

```
do k=1,nz ; do j=1,ny ; do i=1,nx
  Hx(i,j,k) = Hx(i,j,k) + ( (Ey(i,j,k+1)-Ey(i,j ,k)) *Cbdz +      &
                           (Ez(i,j,k  )-Ez(i,j+1,k)) *Cbdy  )
  Hy(i,j,k) = Hy(i,j,k) + ( (Ez(i+1,j,k)-Ez(i,j,k  )) *Cbdx +      &
                           (Ex(i  ,j,k)-Ex(i,j,k+1)) *Cbdz  )
  Hz(i,j,k) = Hz(i,j,k) + ( (Ex(i,j+1,k)-Ex(i  ,j,k)) *Cbdy +      &
                           (Ey(i,j  ,k)-Ey(i+1,j,k)) *Cbdx  )
end do ; end do ; end do
```

where Cbdx etc. are constants. The code for updating the electric field components is very similar.

## References

- [1] Ulf Andersson. Yee\_bench—A PDC benchmark code. TRITA-PDC 2002:1, KTH, November 2002. Available at <http://www.pdc.kth.se/info/research/trita.html>.
- [2] Ulf Andersson, Per Ekman, and Per Öster. Performance and performance counters on the Itanium 2 — A benchmarking case study. In G. R. Joubert et al., editors, *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13 of *Advances in Parallel Computing*, pages 517–524. Elsevier, 2004. Proceedings from ParCo2003.
- [3] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.
- [4] John D. McCalpin. The STREAM2 home page. <http://www.cs.virginia.edu/stream/stream2/>.
- [5] Allen Taflove, editor. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, MA, 1998.
- [6] Allen Taflove. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, MA, third edition, 2005.
- [7] PAPI homepage. <http://icl.cs.utk.edu/papi/>.
- [8] PapiEx homepage. <http://icl.cs.utk.edu/~mucci/papiex/>.
- [9] Finite-Difference Time-Domain literature database. <http://www.fdttd.org/>.