Krzysztof KRASKA, Krzysztof SIEDLECKI
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY, SOFTWARE TECHNOLOGY DEPARTMENT,
Żołnierska 49, 71-210 Szczecin, Poland

Experimental study on data locality of parallel programs executing synchronization-free threads of computations

Ph.D. eng. Krzysztof KRASKA

Ph.D. eng. Krzysztof Kraska is an employee of the Software Technology Department, West Pomeranian University of Technology, Szczecin. His research interest includes parallel computing, optimizing compilers, data locality, and software engineering.

e-mail: kkraska@wi.zut.edu.pl

Ph.D. eng. Krzysztof SIEDLECKI

Ph.D. eng. Krzysztof Siedlecki is an employee of the Software Technology Department, West Pomeranian University of Technology, Szczecin. His research interest includes parallel computing, optimizing compilers, data locality, and software engineering.

e-mail: ksiedlecki@wi.zut.edu.pl

Abstract

The effective use of hierarchical memory for parallel shared memory programs requires good data locality. Analysis and experimental study on data locality in L1D cache for parallel programs executing synchronization-free threads of computations, derived from NAS Parallel Benchmarks, are presented in the paper. Parallel synchronization-free programs were implemented by means of the OpenMP standard. Experiments were carried out in the Intel SMP architecture. The Intel VTune Performance Analyzer was used to collect and evaluate data locality metrics. Finally, a few conclusions about data locality characteristics of synchronization-free parallel programs are given.

Keywords: data locality, synchronization-free parallelism, Intel VTune Performance Analyzer.

Badania eksperymentalne lokalności danych programów wykonujących obliczenia równoległe w niezależnych wątkach

Streszczenie

Efektywne wykorzystanie współczesnych wieloprocesorowych architektur z pamięcią dzieloną, stosujących kilkupoziomową hierarchię dostępu do danych, wymaga od programów wykonujących równolegle obliczenia w niezależnych wątkach dobrych charakterystyk lokalności danych. W niniejszym artykule przedstawiono badania eksperymentalne oraz analizę lokalności danych dla programów zaczerpniętych ze standardowego zestawu testowego NAS Parallel Benchmark, wykonujących obliczenia w niezależnych wątkach utworzonych przy użyciu dyrektyw równoległych standardu OpenMP. Charakterystyki lokalności danych zostały opracowane dla pierwszego poziomu danych (L1D) pamięci cache. Całość badań została wykonana na architekturze Intel SMP z systemem operacyjnym Linux. W celu pozyskania wartości metryk umożliwiających oszacowanie lokalności danych zastosowano narzędzie Intel VTune Performance Analyzer. Na podstawie uzyskanych obserwacji podjęto próbę sformułowania wniosków końcowych.

Słowa kluczowe: lokalność danych, niezależne wątki obliczeń, Intel VTune Performance Analyzer.

1. Introduction

Scientific or engineering applications require many computational cycles to execute the large number of arithmetical operations. A well-known way to speed up computations is to parallelize programs and execute them on multiprocessors. However, to get the effective use of multiple processors, a compiler must be able to detect parallelism in a sequential code and transform the code to expose parallelism. As synchronization on a multiprocessor is very expensive, it is worth extracting synchronization-free threads of computations from the source code. An innovative approach for extracting synchronization-free threads of computations for program loops is presented in [1].

Symmetric multiprocessing (SMP) architectures are most common multiprocessor systems used nowadays. In an SMP system, all processors are connected to a shared memory through the common bus. The bus limits the performance and scalability of the system. To mitigate this disability, a hierarchy of memories is used where each successive memory level is larger in size but has longer access latency. Every processor may have its own *n*-level memory named as *cache*. The highest-level caches are connected to the shared memory through the common bus. Increasing data reuse in each level of the memory hierarchy, especially in the memory closest to a processor, improves *data locality* and can greatly enhance the performance of a program.

Poor data locality is a common feature of many existing numerical programs [2]. Such programs are often represented with affine loops where the considerable quantities of data, placed in arrays, exceed the size of a fast but small cache. In the inefficient code, the referenced data have to be fetched to a cache from a significantly slower memory although they could be reused many times. Improvement in data locality can be achieved by means of high level program transformations. Data locality of a program can be estimated based on the metrics collected from software analysis tools widely available on the market.

The goal of this paper is to present the experimental results of a study on data locality of parallel programs executing synchronization-free threads of computations, and derive conclusions about data locality characteristics of synchronization-free parallel programs.

2. Background

It is said that a program has good *data locality* if a processor often accesses the same data it has used recently. If a program does not have good data locality, its performance can be low.

There are two somewhat different notions of data locality. *Temporal locality* is said to occur when the same data are used several times within a short time period. *Spatial locality* occurs when different data elements that are located near to each other are used within a short time period. An important form of spatial locality occurs when all the elements that appear on one cache line are used together.

Tiling is a well-known technique to improve data locality [3]. If an array cannot fit in a cache, the technique divides the array up into blocks and reorders operations so that an entire block is used over a short period of time.

A *thread* of execution is a sequence of instructions which may execute in parallel with other threads. Threads can exist inside the same process and share resources such as a memory.

The *Hyper-Threading Technology* is an Intel-proprietary technology that enables using execution resources of a processor to execute another task when the processor is stalled.

Synchronization-free slicing [1] is an approach to extract synchronization-free parallelism available in program loops.

3. Parallel NAS Programs

NAS (NASA Advanced Supercomputing) Parallel Benchmarks (NPB) have been developed at the National Aeronautics and Space Administration Ames Research Center to study the performance of parallel supercomputers [4]. We chose pieces of code from NAS Parallel Benchmarks 3.2 Serial Version (SER) for the data locality study. We converted NAS Parallel Benchmarks code from Fortran to C programs. The data layout for arrays in the code was also transposed during conversion because the column-major data layout for arrays is used in the Fortran family of languages while the opposite row-major form is used in C [3]. Next, synchronization-free slices of computations were extracted by means of the method presented in [1] and assigned to parallel threads using OpenMP compiler directives with the default iteration scheduling policy for a compiler. The tiling technique was used for the sequential and the parallel loops to evaluate the impact of a data space size on the data locality of a program. The integer data type was used for all automatic variables and arrays within programs.

FT Benchmark

The FT Benchmark is the Three-Dimensional Fourier Transform. We studied the loop evicted from the subroutine evolve within `auxfcnt.f` file. The index set of the loop was tiled into squares of side `B` for `i`, `j` and `k` indices.

A tiled sequential loop is as follows:

```
#define N1 512
#define N2 512
#define N3 512
#define B 512
...
for(bi=1;bi<=N1;bi+=B)
for(bk=1;bk<=N2;bk+=B)
for(bj=1;bj<=N3;bj+=B) {
for(i=bi;i<(bi+B);i++)
   for(k=bk;k<(bk+B);k++)
      for(j=kj;j<(bj+B);j++) {
         y[i][k][j]*=twiddle[i][k][j];
         x[i][k][j]=y[i][k][j];
      }
}
```

A tiled parallel loop executing synchronization-free threads of computations is as follows:

```
for(bi=1;bi<=N1;bi+=B)
for(bk=1;bk<=N2;bk+=B)
for(bj=1;bj<=N3;bj+=B) {
#pragma omp parallel for firstprivate(bi,bj,bk)
             private(i,j,k) default(shared)
for(i=bi;i<(bi+B);i++)
   for(k=bk;k<(bk+B);k++)
      for(j=bj;j<(bj+B);j++) {
         y[i][k][j]*=twiddle[i][k][j];
         x[i][k][j]=y[i][k][j];
      }
}
```

UA Benchmark

The Unstructured Adaptive (UA) Benchmark measures the effect of irregular, continually changing memory accesses. We studied the loop evicted from the subroutine remap within `adapt.f` file. The index set of the loop was tiled into squares of side `B` for `i`, `ii`, `jj` and `kk` indices.

A tiled sequential loop is as follows:

```
#define N 256
#define B 256
...
for(bi=0;bi<N;bi+=B)
for(bkk=0;bkk<N;bkk+=B)
for(bjj=0;bjj<N;bjj+=B)
for(bii=0;bii<N;bii+=B)
for(i=bi;i<(bi+B);i++) {
  for(kk=bkk;kk<(bkk+B);kk++)
    for(jj=bjj;jj<(bjj+B);jj++)
      for(ii=bii;ii<(bii+B);ii++) {
        yone[0][i][jj][ii]+=ixmc1[kk][ii]*x[i][jj][kk];
        yone[1][i][jj][ii]+=ixmc2[kk][ii]*x[i][jj][kk];
      }
   for(kk=0;kk<(bkk+B);kk++)
    for(jj=0;jj<(bkk+B);jj++)
      for(ii=0;ii<(bkk+B);ii++) {
        ytwo[0][jj][i][ii]+=yone[0][i][kk][ii]*
                            ixtmc1[jj][kk];
        ytwo[1][jj][i][ii]+=yone[0][i][kk][ii]*
                              ixtmc2[jj][kk];
        ytwo[2][jj][i][ii]+=yone[1][i][kk][ii]*
                              ixtmc1[jj][kk];
        ytwo[3][jj][i][ii]+=yone[1][i][kk][ii]*
                              ixtmc2[jj][kk];
      }
}
```

A tiled parallel loop executing synchronization-free threads of computations is as follows:

```
for(bi=0;bi<N;bi+=B)
for(bjj=0;bjj<N;bjj+=B)
for(bii=0;bii<N;bii+=B)
for(bkk=0;bkk<N;bkk+=B)
#pragma omp parallel firstprvate(bi,bii,bjj,bkk)
             private(i,ii,jj,kk) default(shared)
{
  #pragma omp for
  for(i=bi;i<(bi+B);i++)
    for(jj=bjj;jj<(bjj+B);jj++)
      for(ii=bii;ii<(bii+B);ii++)
        for(kk=bkk;kk<(bkk+B);kk++) {
          yone[0][i][jj][ii]+=ixmc1[kk][ii]*x[i][jj][kk];
          yone[1][i][jj][ii]+=ixmc2[kk][ii]*x[i][jj][kk];
        }
  #pragma omp for
  for(i=bi;i<(bi+B);i++)
    for(jj=bjj;jj<(bjj+B);jj++)
      for(ii=bii;ii<(bii+B);ii++)
        for(kk=bkk;kk<(bkk+B);kk++) {
          ytwo[0][jj][i][ii]+=yone[0][i][kk][ii]
                              *ixtmc1[jj][kk];
          ytwo[1][jj][i][ii]+=yone[0][i][kk][ii]
                              *ixtmc2[jj][kk];
          ytwo[2][jj][i][ii]+=yone[1][i][kk][ii]
                              *ixtmc1[jj][kk];
          ytwo[3][jj][i][ii]+=yone[1][i][kk][ii]
                              *ixtmc2[jj][kk];
        }
}
```

4. Experiments

The data locality study on the parallel programs was performed based on the first-level data (L1D) cache. The number of threads in OpenMP parallel regions was left at the default compiler value, i.e. 32 threads were created to execute parallel operations. Compiler optimizations were disabled to exclude the impact of any other optimization technique on data locality. Side `B` of tiles was manipulated to be a multiplicity of the L1D Cache Line Size.

Experiments were performed on the IBM System x3950M2 with 8 x Quad Core Intel Xeon Processor E7310 (in total 32 cores). The CPU parameters relevant to the experimental results are as follows (based on the Linux `dmidecode` command):

a) number of physical cores: 4,
b) core L1D cache size: 128kB,
c) core L1D cache line size: 64B,
d) L1D Cache associativity: 8-way set,
e) L1D Cache operational mode: Write Back,
f) Intel Hyper-Threading Technology: No.

The examined sources were compiled by means of the Intel C++ Compiler v11.0 with disabled optimizations (`-Od`), enabled generation of the multi-threaded OpenMP code, and executed on Linux openSUSE v11.1. The size of the `int` type is 4-bytes.

In order to reduce the impact of unwanted factors on the obtained results, no other programs were executed during the experiments. However, there was no possibility to exclude the impact of the operating system software running in the background.

The data locality metrics were collected using the Intel VTune Performance Analyzer v9.1 for Linux.

Contemporary Intel processors enable monitoring the performance events by means of the built-in Performance Management Unit (PMU). To evaluate the data locality of parallel programs, the following PMU metrics were collected (complete explanation is given in [5]):

a) `L1D_ALL_CACHE_REF` – counts the number of data reads and writes from L1D cache memory, including locked operations,
b) `L1D_CACHE_LD.MESI` – counts how many times L1D cache lines in any state are accessed for data reading,
c) `L1D_CACHE_ST.MESI` – counts how many times L1D cache lines in any state are accessed for data writing,
d) `L1D_PEND_MISS` – counts the number of outstanding L1D cache misses at any cycle; an L1D cache miss is outstanding from the cycle on which the miss is determined until the first chunk of data is available,
e) `L1D_REPL` – counts the number of lines brought into the L1D cache,
f) `L1D_M_EVICT` – counts the number of modified lines evicted from the L1D cache, whether due to replacement or by snoop HITM intervention,
g) `MEM_LOAD_RETIRED.L1D_LINE_MISS` – counts the number of load operations that miss the L1D cache and sends a request to the L2 cache to fetch the missing cache line.

The results of experiments performed on the studied programs are presented in Figs.1 and 2. In the figures:

- the curve with squares shows the number of events for a sequential program,
- the curve with circles shows the total number of events combined from all cores for a parallel program executing synchronization-free threads of computations,
- the curve with diamonds shows the average number of events on one processor for a parallel program executing synchronization-free threads of computations.

The results are presented in the form of an interpolated curve only to show a general trend of changes.

FT Benchmark

Collections of the L1D cache PMU metrics for the loop derived from the FT Benchmark are presented in Fig. 1.

Inside the loops, the three arrays `y`, `twiddle` and `x` are processed. Each array has 512^3 `int` type elements that gives 512MB per array and 1,5GB total amount of memory allocated for all arrays. Side `B` of a tile was multiplied by 2^n*16 `int` type elements (2^n*64 bytes), where n={1,2,3,4}.

The results for both sequential and parallel loop without tiling are shown for `B` = 512. At this point remarks about the results of running parallel synchronization-free threads of computations can be concluded. A consequence of using the tiling technique for different values of `B` on the sequential loop is represented by the shape of the curve with squares. The shape of the curve with circles shows the consequences of applying both: (1) synchronization-free slicing and (2) the tiling technique with different values of `B`. For the parallel loop executing synchronization-free threads of computations the events were decomposed in roughly evenly (about 1/32) on each core during experiments.

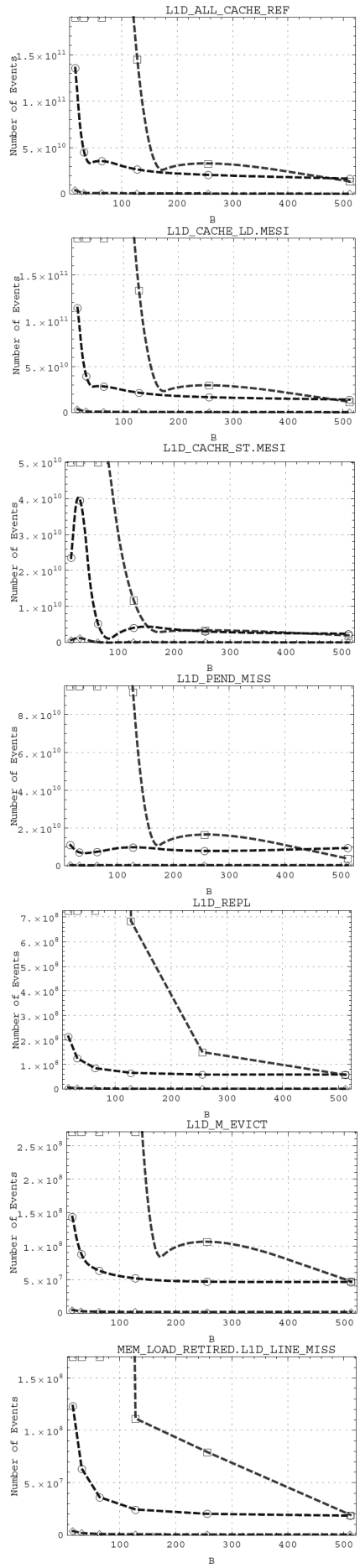

Fig. 1. Data locality metrics for a loop from the FT Benchmark
Rys. 1. Metryki lokalności danych dla pętli z FT Benchmark

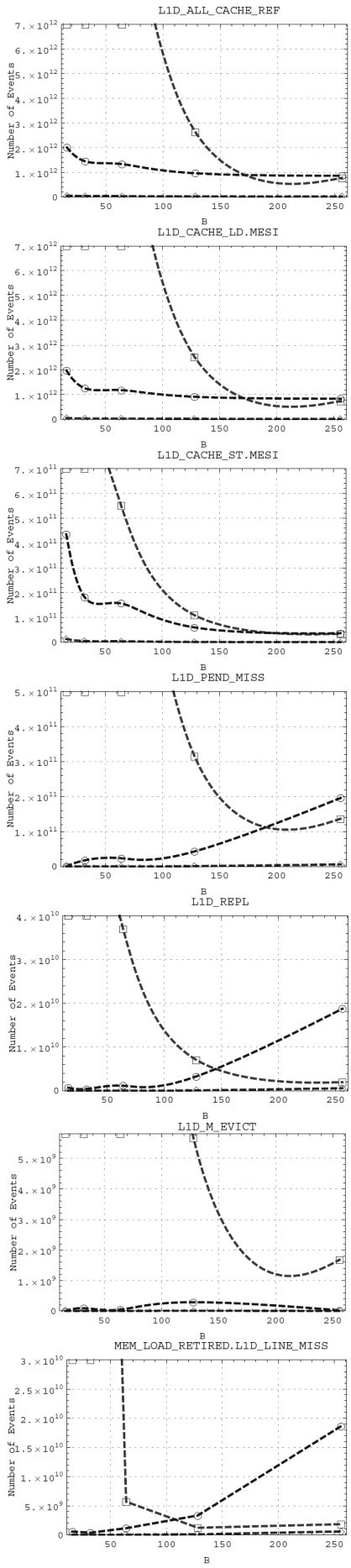

Fig. 2. Data locality metrics for a loop from the UA Benchmark
Rys. 2. Metryki lokalności danych dla pętli z UA Benchmark

UA Benchmark

Collections of the L1D cache PMU metrics for the loop derived from the UA Benchmark are shown in Fig. 2.

Inside the loops, the seven arrays `yone` ($2*256^3$ `int` type elements give 128MB), `ytwo` ($4*256^3$ `int` type elements give 256MB), `ixmc1` (256^2 `int` type elements give 256kB), `ixmc2` (256^2 `int` type elements give 256kB), `ixtmc1` (256^2 `int` type elements give 256kB), `ixtmc2` (256^2 `int` type elements give 256kB) and `x` (256^3 `int` type elements give 64MB) are processed. The total memory allocated for all arrays equals 449MB. Side `B` of a tile was multiplied by 2^n*16 `int` type elements (2^n*64 bytes), where n={1,2,3}.

Within the innermost sequential loops an access to the arrays `yone`, `ytwo`, `ixmc1`, `ixmc2` is made in continuous memory space dimensions while the index expressions of the arrays `x`, `ixtmc1`, `ixtmc2` are constant (incremented by the outer loops). The synchronization-free slicing causes loop permutation and creates the opposite situation – within the innermost loops the arrays `x`, `ixtmc1`, `ixtmc2` are accessed in continuous memory space dimensions, while the index expressions of the arrays `yone`, `ytwo`, `ixmc1`, `ixmc2` are constant.

The results for both sequential and parallel loop without tiling are shown for `B` = 256. At this point remarks about the results of running synchronization-free threads of computations in parallel can be concluded. A consequence of using the tiling technique for different values of `B` on the sequential loop is represented by the shape of the curve with squares. The shape of the curve with circles shows consequences of applying both: (1) synchronization-free slicing and (2) the tiling technique with different values of `B`. Like the FT Benchmark, for the parallel loop executing synchronization-free threads of computations the events were decomposed in roughly evenly (about 1/32) on each core during experiments.

5. Conclusions

The following remarks can be drawn from the results of the performed study on the data locality characteristics of synchronization-free parallel programs.

1) The parallel programs executing synchronization-free threads of computations in the multiprocessor environment use a combined L1D cache memory (32 cores * 128kB gives 4MB) and distribute a data space among all cores. Sequential versions of the examined programs are executed on a single core with only 128kB L1D cache. Hence, the obvious conclusion is that slicing compared to sequential processing increases utilization of L1D cache resources available in a multiprocessor environment.
2) Even though the UA Benchmark processes less data space than the FT Benchmark, it is more memory intensive and, hence, results in more L1D cache events. There are more references to the data in the UA Benchmark code. The conclusion is that each program has different characteristics like the FT Benchmark and the UA Benchmark.
3) Tiling is commonly recognized as an effective technique to improve data locality [3]. In the experiments a smaller size of the tile intensified use of the L1D cache (as shown by `L1D_ALL_CACHE_REF`, `L1D_CACHE_LD.MESI` and `L1D_CACHE_ST.MESI` metrics) both in sequential and parallel programs. But, at the same time, it increased the number of L1D cache misses (as shown by `L1D_PEND_MISS`, `L1D_REPL`, `MEM_LOAD_RETIRED.L1D_LINE_MISS`, `L1D_M_EVICT` metrics) for sequential programs. We tiled the continuous memory space and, hence, the data locality metrics, unfortunately, are worse. It is the same for the parallel FT Benchmark, but less memory intensive characteristics of the program and larger L1D cache resources mitigated the impact of the unfavorable factor. However, the memory intensive parallel UA Benchmark has the opposite feature – a smaller size of the tile reduces the number of the

L1D cache misses. The conclusion is that unreasonable use of tiling can, contrary to the intentions, decrease the performance of a program and the tiling of a data space in programs is not obvious at the first sight for developers, so the reliable analysis supported by tools is needed.

4) Manufacturer's recommendation that block sizes of all tiles should target approximately one-half to three-quarters of the size of the physical cache size for a processor without the Hyper-Threading technology and one-quarter to one-half of the physical cache size for a processor equipped with the Hyper-Threading Technology [6] is not accurate enough. It is necessary to tune-up block sizes of tiles for every program to achieve better data locality.

In our future research, we are going to implement a module of an automatic parallelizing and optimizing source-to-source compiler. The module will automatically perform data locality optimization tailoring parallel synchronization-free threads of computations to a target architecture specification based on the L1D and L2D cache metrics.

6. References

[1] Bielecki W., Beletska A., Siedlecki K., San Pietro P.: Finding Synchronization-Free Slices of Operations in Arbitrarily Nested Loops. In: ICCSA 2008, LNCS, vol. 5073, Springer, 2008.
[2] Griebl M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. Habilitation. Universitat Passau, 2004.
[3] Aho A., Lam M., Sethi R., Ullman J.: Compilers: Principles, Techniques and Tools, 2nd Editio. Pearson Higher Education, 2007.
[4] NASA Advanced Supercomputing Parallel Benchmarks Version 3.2, http://www.nas.nasa.gov/Software/NPB/
[5] Intel VTune Performance Analyzer, VTune Performance Environment Help. Intel Corporation, 2008.
[6] Threading Methodology: Principles and Practices. Version 2.0. Intel Corporation, 2004.

otrzymano / received: 10.09.2010
przyjęto do druku / accepted: 01.11.2010 *artykuł recenzowany*

INFORMACJE