Caches are intermediate level between fast CPU and slow main memory. It aims to store copies of frequently used data and to reduce the access time to the main memory. Caches are capable of exploiting temporal and spatial localities during program execution. When the processor accesses memory, the cache behavior depends on if the data is in cache: a cache hit occurs if it is, and a cache miss occurs, otherwise. In the last case, the cache may have to evict other data. The misses produce processor stalls and slow down the computations. The replacement policy chooses a data to evict, trying to predict the future accesses to memory. The hit and miss rate depends on the cache type: direct mapped, set associative and fully associative cache. The least recently used replacement policy serves the sets. The miss rate strongly depends on the executed algorithm. The all pairs shortest paths algorithms solve many practical problems, and it is important to know what algorithm and what cache type match best. This paper presents a technique of simulating the direct mapped, k-way associative and fully associative cache during the algorithm execution, to measure the frequency of read data to cache and write data to memory operations. We have measured the frequencies versus the cache size, the data block size, the amount of processed data, the type of cache, and the type of algorithm. After comparing the basic and blocked Floyd-Warshall algorithms, we conclude that the blocked algorithm well localizes data accesses within one block, but it does not localize data dependencies among blocks. The direct mapped cache significantly loses the associative cache: we can improve its performance by appropriate mapping virtual addresses to physical locations.

Keywords: hierarchical memory, direct mapped cache, k-way associative cache, fully associative cache, all pairs shortest paths algorithms, performance, simulation.

Introduction

Caches are intermediate level between CPU and main memory, which reduces the average time and energy to access data stored in the main memory. Caches keep copies of data from frequently used locations of the memory. Most modern CPUs have three caches [1-3]: an instruction cache, a data cache, and a translation lookaside buffer. The data cache is usually a hierarchy of some cache levels. In a multi-core processor, the lower levels of cache hierarchy are split among cores, and the higher cache levels act as a common repository of data for all cores.

The data is transferred between the main memory and the cache in blocks (lines). When the processor reads or writes a memory location, the cache checks if the line is in cache. If the cache reads a line, it creates a cache entry, which includes the copied data and the memory location (called a tag). If the location is in the cache, a cache hit has occurred; otherwise, a cache miss has occurred. As CPUs are much faster than the memory, stalls due to the cache misses slow down the computation significantly. The key step in improving the cache performance is reducing the miss rate.

To prepare a cache slot on a cache miss to read the requested entry, the cache may have to evict one of the existing entries. The replacement policy chooses an entry to evict. It tries to predict the future accesses to the entries in cache. One of the most popular and efficient replacement policies is LRU that replaces the least recently used entry. At some point, the cache must write the updated data to memory. Two write policies can do this: the first one known as “write-through cache” performs the write to memory with every write to cache; the second one known as “write- back cache” tracks by means of a dirty bit which loca-
tions have to be written (it writes the dirty data to memory only when replaces it with other data).

In the recent times, the cache performance measurements help in bridging the gap in the speed of the processor and memory in high-performance computing systems. The cache performance significantly depends on what algorithm the processor runs. This paper investigates how the type (direct mapped, k-way or fully associative) of cache \[1\] influences the algorithm runtime, and how we can modify the algorithms to obtain the increased performance of the cache (to do this we need to obtain the reduced number of cache read and write operations). In this work, we focus on the simulation and analysis of sequential algorithms in relation to properties of various cache types; therefore, the emphasis of the paper is on the one-core-processor-cache-memory architecture.

**Organization of caches**

In cache, there are three placement options for where data can go: direct-mapped, fully associative, and set-associative. The k-way associative cache organization (it maps each memory line to a subset of cache slots) is set up to exploit temporal (if accessed, will access again soon) and spatial (if accessed, will access others around it) locality. Figure 1 shows the mechanism of mapping memory lines to cache slots at \(Nset = 2\) and \(Kway = 2\). Lines 0, 2, 4… of memory can be assigned to any of two ways of set 0, and lines 1, 3, 5… can be assigned to ways of set 1.

If \(Nset = 1\), the cache becomes the fully associative cache: the incoming tag must be compared with all cache tags as the cache maps each memory line to any cache slot. If \(Kway = 1\), the cache becomes the direct mapped cache: the incoming address tag must be compared with only one cache tag as the cache maps each memory line to exactly one cache slot.

The method by Maruyama (Figure 2) implements the real LRU for each of \(Nset\) cache sets. It keeps one matrix \([Kway\times Kway]\) of bits for each set. When line \(i\) of a set accessed, the method per-

![Figure 1. Mapping memory lines to cache slots in 2-way associative cache](image-url)

![Figure 2. Maruyama method for real LRU on one set](image-url)
forms steps as follows: (1) set row \( i \) to all ones; (2) set column \( i \) to all zeroes; (3) evicted line corresponds to all zero row. For example, line accesses are made in the order 2-1-0-3-1-2. Figures 2a–2f show six states of the bit matrix that indicate by their zero rows the evicted slots: \( \{0, 1, 3\}, \{0, 3\}, \{3\}, \{2\}, \{2\}, \{0\} \). In our work, we implement the “write-back cache” policy [1].

**Floyd–Warshall algorithm**

Let’s consider a directed graph \( G = (V, E) \), where \( V = \{0, \ldots, N - 1\} \) and \( E \subseteq \{(i, j) \mid i, j \in V\} \) are the vertex and edge sets, respectively. A function \( w : E \rightarrow R \) assigns the weight \( w_{ij} \) to edge \((i, j) \in E \). Matrix \( W \) represents the function, in which \( W(i, j) = 0 \) if \( i = j \), \( W(i, j) = w_{ij} \) if \((i, j) \in E \), and \( W(i, j) = \infty \) if \((i, j) \notin E \).

The all-pairs shortest paths problem is formulated as to find the paths of the shortest length for all pairs of vertices \( i, j \in V \). Algorithm 1 known as Floyd–Warshall (FW) algorithm [4], uses a matrix \( D \) that describes the all pairs shortest paths lengths. The loop iterations on \( k \) produce the states \( D^0, D^1, D^2, \ldots, D^N \) of \( D \) according to the recurrent equation as follows:

**Algorithm 1:** Floyd–Warshall (FW)

*Input:* A number \( N \) of graph vertices

*Input:* A matrix \( W \) of graph edge weights

*Output:* A matrix \( D \) of all-pairs shortest paths lengths

\[ D \leftarrow W \]

for \( k \leftarrow 0 \) to \( N - 1 \) do

for \( i \leftarrow 0 \) to \( N - 1 \) do

for \( j \leftarrow 0 \) to \( N - 1 \) do

\[ D_{ij}^{k+1} = \min\{D_{ij}^{k}, D_{ik}^{k} + D_{kj}^{k}\} \]

return \( D \)

\[ D_{ij}^{k+1} = \min\{D_{ij}^{k}, D_{ik}^{k} + D_{kj}^{k}\} \tag{1} \]

State matrix \( D^0 = W \), and state matrix \( D^N \) describes the final shortest paths lengths. The computational complexity of algorithm FW is \( O(N^3) \). For large matrices, algorithm FW can consume a lot of execution time, the significant part of which is due to the operations in the hierarchical memory.

**Blocked Floyd–Warshall algorithm**

Let the \( N \times N \) matrix \( D \) be blocked into an \( M \times M \) matrix of smaller matrices \( B_{ij} \), \( 0 \leq j, j \geq B \), where \( B = N / M \). Algorithm 2 known as the blocked Floyd–Warshall (BFW) algorithm [5–6], iteratively calls a function \( \text{Block}(B^1, B^2, B^3) \) of recalculating block \( B^1 \) over blocks \( B^2 \) and \( B^3 \) (Algorithm 3).

**Algorithm 2:** Blocked Floyd–Warshall (BFW)

*Input:* A number \( N \) of graph vertices

*Input:* A matrix \( W \) of graph edge weights

*Output:* A matrix \( D \) of lengths of all pairs shortest paths

\[ M \leftarrow N / B \]

\[ D[M \times M] \leftarrow W[N \times N] \]

for \( m \leftarrow 0 \) to \( M - 1 \) do

\[ \text{Block}(B_{m,m}, B_{m,m}, B_{m,m}) \] // \( D \)

for \( i \leftarrow 0 \) to \( M - 1 \) do

if \( i \neq m \) then

\[ \text{Block}(B_{i,m}, B_{i,m}, B_{m,i}) \] // \( C \)

for \( i \leftarrow 0 \) to \( M - 1 \) do

if \( i \neq m \) then

\[ \text{Block}(B_{i,j}, B_{i,m}, B_{m,j}) \] // \( U \)

return \( D \)

**Algorithm 3:** Recalculation of block

*Input:* \( B \) – size of block

*Input:* \( B^1 \) – first input block

*Input:* \( B^2 \) – second input block

*Input:* \( B^3 \) – third input block

*Output:* \( B^1 \) – recalculated block

for \( m \leftarrow 0 \) to \( B - 1 \) do

for \( j \leftarrow 0 \) to \( B - 1 \) do

\[ \text{sum} \leftarrow B_{i,j} + B_{i,m} \]

if \( B^1 > \text{sum} \) then

\[ B^1 \leftarrow \text{sum} \]

return \( B^1 \)

Figure 3 illustrates the behavior of BFW on the matrix \([4 \times 4]\) of blocks. In the first iteration, BFW recalculates diagonal (D) block \( B_{00} \), recalculates blocks of the cross (C) with the center in \( B_{00} \), and then recalculates other blocks (U). In the second iteration, the cross moves to block \( B_{11} \), in the third iteration it moves to \( B_{22} \), and so on.

The computational complexity of BFW is the same as that of FW, but in contrast to FW, BFW can localize data and computations within the block, which is very important for caches, and it
Simulation of algorithms FW and BFW on caches

Simulation is an efficient technique to measure dynamic parameters of a complex system represented as a computer program [7–8] at the behavioral level.

Algorithm 4 describes the cache-based simulation of algorithm FW. It aims at measuring the number of read and write operations in the cache occurred during execution of FW, and extends Algorithm 1 in the following points:

- initializing the cache model by zeroing the line read and write counters, and initializing arrays depending on the cache type,
- calculating the memory line number of \( L_{ik} \), \( L_{kj} \), and \( L_{ij} \) depending on the elements of matrix \( D \),
- simulating the read-write operations and the line miss over function \( \text{MemoryAccess}(L) \) that is implemented depending on the cache type,
- simulating the write of a line to memory.

We organize the cache-based simulation of algorithm BFW in a similar way. Since the computer memory is inherently linear, algorithm FW uses the row-major memory layout of matrix \( D \). Algorithm BFW uses the block-major memory layout of whole matrix \( D \), and uses the row-major layout of each block. For the line representation of memory, Table 1 reports the number of lines in one block, in the whole matrix \( D \), and in the cache depending on the line size. The modules of cache run mostly in parallel. At the same time, our cache simulation program operates sequentially. Therefore, the model of cache simulation slightly differs from the real cache model.

**Table 1. Number of lines in block [8×8], in matrix \( D[64×64] \), and the number of slots in cache of 1024 byte vs. line size**

<table>
<thead>
<tr>
<th>Line size, bit</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block lines</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Matrix lines</td>
<td>2048</td>
<td>1024</td>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>Cache slots</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

**Algorithm 4:** Simulation of algorithm FW for cache

**Input:** A number \( N \) of vertices in graph
**Input:** A matrix \( W \) of graph edge weights
**Input:** A size \( L_{size} \) in words of cache line
**Input:** A number \( C_{size} \) of lines in cache
**Input:** Variables of a cache model

---

Simulation of direct mapped cache

We model the direct mapped cache at abstract level using the variables as follows:

- \( \text{Tag} \) is an array of size \( C_{size} \), which elements are addresses of memory lines read in cache.
- \( \text{DirtyC} \) is an array of size \( C_{size} \), which elements are flags indicating the cache lines updated.

**Initialization of direct mapped cache:**

\[
\text{read} ← 0; \quad \text{write} ← 0;
\]

for \( i ← 0 \) to \( C_{size}-1 \) do

\[
\text{Tag}[i] ← 1;
\]

\[
\text{DirtyC}[i] ← \text{false}
\]

**Simulating the write of line to memory:**

\[
\text{DirtyC}[L_{ij} \% C_{size}] ← \text{true}
\]

**Destruction of the cache model:**

for \( i ← 0 \) to \( C_{size}-1 \) do

if \( \text{DirtyC}[i] \) then ++ write

Algorithm 5 implements the function \( \text{MemoryAccess}(L) \) and models a cache miss. It first calculates the value of \( \text{index} \) and \( \text{tag} \) for line \( L \). Variable \( \text{index} \) indicates the cache slot that holds \( L \). When \( L \) is not in cache, \( \text{Tag}[\text{index}] \neq \text{tag} \). If the current data of cache slot \( \text{index} \) is dirty, the algorithm writes the data to line \( \text{Tag}[\text{index}] \times C_{size} + \text{index} \) and assigns \( \text{false} \) to \( \text{DirtyC}[\text{index}] \). Then it reads line \( L \) in cache and assigns the tag of \( L \) to \( \text{Tag}[\text{index}] \).

**Algorithm 5:** Simulation of memory access in direct mapped cache

**Input:** A line \( L \) of memory
**Input:** A number \( C_{size} \) of lines in cache
**InOut:** An array \( \text{Tag} \) of memory line tags that are in cache
**InOut:** An array \( \text{DirtyC} \) of updated cache lines
**InOut:** A number \( \text{read} \) of memory line reads in cache
Simulation of k-way associative cache

Let $Dsize$ be the number of lines allocated for matrix $D$. We model the k-way associative cache at abstract level using the variables as follows:

- **InCache** is an array of size $Dsize$, which elements are flags indicating $D$-matrix (memory) lines read in cache,
- **DirtyM** is an array of size $Dsize$, which Boolean elements indicate the memory lines dirty in cache,
- **Valid** is an array of size $Csize$, which Boolean elements indicate the valid data in cache slots,
- **BitsL** is an array of size $Csize$, which elements represent the rows of $K^2$-matrices of bits assigned to the cache sets.

Initialization of k-way associative cache:

```plaintext
read ← 0
write ← 0
for $j ← 0$ to $Dsize-1$ do
  InCache[$j$] ← false
  DirtyM[$j$] ← false
for $i ← 0$ to $Csize-1$
  Tag[$i$] ← read
  Valid[$i$] ← false
  BitsL[$i$] ← 0
```

Simulating the write of line to memory:

```plaintext
DirtyM[Tag] ← true.
```

Algorithm 6 describes the procedure of simulating the cache miss in the k-way associative cache. It calculates tag and index of line $L$ and implements the Maruyama and write-back methods, which do not write the dirty data to memory until necessary. Variable $displ$ indicates the first slot of the cache set that accommodates line $L$. When the first loop breaks, $sl$ indicates either a free cache slot for reading line $L$, or indicates a cache slot that already holds the data of $L$. Variable $ul$ indicates a bit-matrix row in the array $BitsL$, which takes the value of $mask[sl]$ that is a sequence of length $Kway$ of ones except element $sl$ that is zero. This value also updates all bit-matrix rows of the set according to the Maruyama method, by means of Boolean operation and on bit-vectors. If line $L$ is not in cache, $Lr$ denotes a line that is currently in the selected slot $ul$. If the slot holds valid-dirty data, the algorithm increments the value of counter write, resets the dirty bit, and marks line $Lr$ as out of cache. After that, it marks line $L$ as read in the cache, and increments the value of counter read.

Algorithm 6: Simulation of memory access in k-way associative cache

```plaintext
Input: A line $L$ of memory
Input: A number $Nset$ of sets in cache
Input: A number $Kway$ of slots in one set
Input: An array $BitsS$ of sample bit-vectors
Input: An array $InCache$ of flags indicating read memory lines
Input: An array $DirtyM$ of flags indicating updated memory lines
Input: An array $Valid$ of flags indicating valid data in cache slots
```

Simulation of fully associative cache

For fully associative cache, we simulate the replacement strategy LRU that serves all cache slots. We implement LRU in a way different to the Maruyama method. The simulation procedure works at abstract level using the $Tag$, $Valid$ and $DirtyC$ arrays, and additionally using the variables as follows:

- **Time** is a counter of time points,
- **Slot** is an array of size $Dsize$, which elements are cache slot indices assigned to lines,
- **Rtime** is an array of size $Csize$, which elements are time points of referring to lines held in cache.

Fully associative cache initialization:

```plaintext
read ← 0; write ← 0; Time ← 0;
for $j ← 0$ to $Dsize-1$
  Slot[$j$] ← −1.
```
for \( i \leftarrow 0 \) to \( \text{Csize} - 1 \) do
\[
\text{Tag}[i] \leftarrow -1 \quad \text{Rtime}[i] \leftarrow -1 \\
\text{Valid}[i] \leftarrow \text{false} \quad \text{DirtyC}[i] \leftarrow \text{false}
\]

Simulating the write of line to memory:
\[
\text{DirtyC}[\text{Slot}[\text{loc}]] \leftarrow \text{true}
\]

Algorithm 7 describes the procedure of simulating the memory line miss in the fully associative cache. Variable \( sl \) indicates the cache slot that holds the memory line \( L \). If \( sl \neq -1 \), line \( L \) is in cache, variable \( \text{Rtime}[sl] \) gets the value of \( \text{Time} \), and the algorithm returns the control. Otherwise, in a loop it searches for a slot \( \text{loc} \) of cache for accommodating the line \( L \). The slot either contains a garbage or is a least recently used one.

**Algorithm 7: Simulation of memory access in fully associative cache**

- **Input:** A line \( L \) of memory
- **Input:** A number \( \text{Csize} \) of lines in cache
- **InOut:** A counter \( \text{Time} \)
- **InOut:** An array \( \text{Tag} \) of memory line tags that are in cache
- **InOut:** An array \( \text{Valid} \) of flags indicating valid data in cache slots
- **InOut:** An array \( \text{DirtyC} \) of slots with updated data
- **InOut:** An array \( \text{Slot} \) of cache slots assigned to memory lines
- **InOut:** An array \( \text{Rtime} \) of time points of reference to data in slots
- **InOut:** A number \( \text{read} \) of memory line loads in cache
- **InOut:** A number \( \text{write} \) of data-in-slot writes to memory

\[
\text{sl} \leftarrow \text{Slot}[L] + \text{Time} \\
\text{if} \; \text{sl} \neq -1 \; \text{then} \\
\quad \text{Rtime}[sl] \leftarrow \text{Time} \quad \text{return} \\
\quad \text{tmin} \leftarrow \text{Time} \\
\quad \text{for} \; \text{cl} \leftarrow 0 \; \text{to} \; \text{Csize} - 1 \; \text{do} \\
\quad \text{if} \; \text{not} \; \text{Valid}[cl] \; \text{then} \\
\quad \quad \text{loc} \leftarrow \text{cl} \; \text{break} \\
\quad \text{if} \; \text{tmin} > \text{Rtime}[cl] \; \text{then} \\
\quad \quad \text{tmin} \leftarrow \text{Rtime}[cl] \; \text{loc} \leftarrow \text{cl} \\
\text{if} \; \text{Valid}[loc] \; \text{then} \\
\quad \text{if} \; \text{DirtyC}[loc] \; \text{then} \\
\quad \quad ++\text{write DirtyC}[loc] \leftarrow \text{false} \\
\quad \text{Slot}[\text{Tag}[loc]] \leftarrow -1 \\
\text{Tag}[loc] \leftarrow L \quad \text{Rtime}[loc] \leftarrow \text{Time} \\
\text{Slot}[L] \leftarrow \text{loc} + \text{read} \\
\]

If slot \( \text{loc} \) contains a valid-dirty data, the algorithm increments the counter \( \text{read} \), resets the flag \( \text{DirtyC}[loc] \), and sets the value of \( \text{Slot}[\text{Tag}[loc]] \) to \(-1\). Finally, it reads line \( L \) in cache, sets the line reference time to \( \text{Time} \), and fixes the cache slot of line \( L \) to be \( \text{loc} \). The procedures of simulating FW and BFW increment the value of counter \( \text{Time} \).

**Experimental results**

This section compares FW and BFW algorithms, regarding the number of read and write operations in each of three cache types: direct mapped, \( k \)-way associative, and fully associative. It also studies the algorithm features while increasing the size of matrix \( D \). We performed experiments on the same for algorithms and for caches randomly generated weighted complete graphs at various line, block and graph size. Comparison of the algorithms and caches for various line size. Matrix \( D \) of \( 64 \times 64 \) elements of 4 byte each requires totally 16384 byte of memory. One block of \( 8 \times 8 \) elements of 4 byte each occupies 256 byte of memory. Matrix \( D \) consists of \( 8 \times 8 \) blocks.

In the experiments, we use caches of two sizes 1024 and 512 byte. The first size cache can hold four blocks, which is larger than three input blocks of function \( Cblock \). The second size cache can hold only two blocks, and cannot accommodate all data the function \( Cblock \) needs. That is why this size is flaky and can produce many read operations. The line size varies in the range from 8 to 256 byte, therefore the caches can accommodate from 2 to 128 lines.

Figure 4 presents results for read operations in the direct mapped cache. The larger the line size the lower the number of read operations for FW and the larger the number of read operations for BFW. At a low line size, BFW has a minimum of reads (111767), but it loses to FW significantly at the high line size.

Figure 5 reports results for the 2-way associative cache. The behavior of curves is very similar to that in Figure 4. A distinction is FW yields fewer read operations against BFW. For 4-way associative cache, the situation has dramatically changed (Figure 6). BFW overcomes FW at any line size, having a minimum of the line read operations (959).

Figure 7 presents results for the fully associative cache. FW at the line size of 512 and 1024,
and BFW at the line size of 1024 have given the results that are very close to that obtained for 4-way associative cache. The results distinct only for BFW at the line size of 512. This is due to the 512 byte cache cannot fit three blocks. We can conclude that $k$-way associative cache approaches to fully associative cache very rapidly with increasing $k$.

Now we compare the algorithms and caches regarding write operations on dirty lines. Figures 8–11 show the number of write operations versus the line size for two algorithms and three caches. In all caches, the number decreases for FW. BFW gives a larger number of write operations for the direct mapped and for the 2-way associative cache. For the 4-way and fully associative cache, the number of write operations falls, and the gain of BFW over FW is significant.

**Comparison of FW and BFW while scaling the problem size.** We explore the fully associative cache to find out how the increase in the size of matrix $D$ influences the features of FW and BFW. For the matrix size from 4 to 36 times larger to the cache size, the reduction in number of line reads produced by BFW slightly exceeds 4 times against FW (Figure 12). When the matrix size grows from 64 to 121 times, the reduction reaches 8.79 times. For larger matrix size when the matrix’s row size is equal to the size of three blocks, the reduction rapidly falls to 1.0, which means BFW has no advantages to FW regarding the us-
age of caches for solving very large size problems. We can explain this as BFW localizes accesses to lines within one block, but it does not localize data dependencies among blocks.

The reduction in the number of write operations in BFW against FW monotonically falls from 6.22 down to 1.45 times when the matrix size grows from 4 to 256 times against the cache size (Figure 12). This is a significant advantage of BFW.

**Conclusion**

We have developed the abstract-level simulation technique and tool, which allow the measurement of performance parameters of various type of caches during execution of important algorithms. These help us in the comparison of caches and in the comparison of alternative algorithmic implementations for solving the same problem. In particular, we can conclude that the direct mapped cache significantly loses to the $k$-way and fully associative cache with respect to the number of read and write operations executed while solving the all pairs shortest paths problem. We also conclude that the blocked Floyd–Warshall algorithm overcomes the basic Floyd–Warshall algorithm in the efficiency of cache operation, but the blocked algorithm needs to be improved for very large graphs.

**REFERENCES**


Поступила 14.10.2019
После доработки 29.11.2019
Принята к печати 01.12.2019

ПРИХОЖИЙ А.А.

МОДЕЛИРОВАНИЕ КЭШ ПРЯМОГО ОТОБРАЖЕНИЯ И АССОЦИАТИВНЫХ КЭШ НА АЛГОРИТМАХ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ В ГРАФЕ

Кэш является промежуточным уровнем между быстрым процессором и медленной основной памятью. Он предназначен для хранения копий часто используемых данных и сокращения времени доступа к основной памяти.
Кэш способен использовать временную и пространственную локальность данных во время выполнения программы. Когда процессор обращается к памяти, поведение кэш зависит от того, находятся ли данные в нем: попадание в кэш происходит, если данные там, в противном случае, имеет место промах кэш. В последнем случае кэш может потребоваться удалить другие данные. Промахи приводят к остановке процессора и замедляют вычисления. Для снижения частоты промахов в кэше выбирают данные для удаления, пытаясь предсказать будущие обращения к памяти. Частота промахов зависит от типа кэш: прямого сопоставления, множественно-ассоциативного и полностью ассоциативного кэш. Стратегии удаления наименее недавно использованных данных обслуживают множества слотов. Уровень промахов сильно зависит от выполняемого алгоритма. Алгоритмы поиска кратчайших путей между всеми вершинами графа решают многие практически задачи, важно знать, какой алгоритм и какой тип кэш лучше подходят друг другу. В этой статье представлен метод моделирования кэш прямого отображения, k-канального ассоциативного и полностью ассоциативного кэш во время выполнения алгоритма, для измерения частоты чтения данных из кэш и записи данных в память. Мы измерили частоты в зависимости от размера кэш, размера блока данных, объема обработанных данных, типа кэш и типа алгоритма. После сравнения основного и блочного алгоритма Флойда-Уоршелла, мы пришли к выводу, что блочный алгоритм хорошо локализует доступ к данным внутри одного блока, но не локализует зависимости данных между блоками. Кэш прямого отображения значительно уступает ассоциативным кэш; мы можем улучшить его производительность путем соответствующего отображения виртуальных адресов на физические адреса памяти.

Ключевые слова: иерархическая память, кэш прямого отображения, k-канальный ассоциативный кэш, полностью ассоциативный кэш, задача поиска кратчайших путей, алгоритмы поиска, производительность, имитационное моделирование.

Anatoly Prihozhy is a full professor at the Computer and system software department of Belarusian national technical university, doctor of science (1999) and full professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design techniques and tools for software and hardware at logic, high and system levels, and for incompletely specified logical systems. He has over 300 publications in Eastern and Western Europe, USA and Canada. Such worldwide publishers as IEEE, Springer, Kluwer Academic Publishers, World Scientific and others have published his works.