Организация памяти

Иерархии памяти

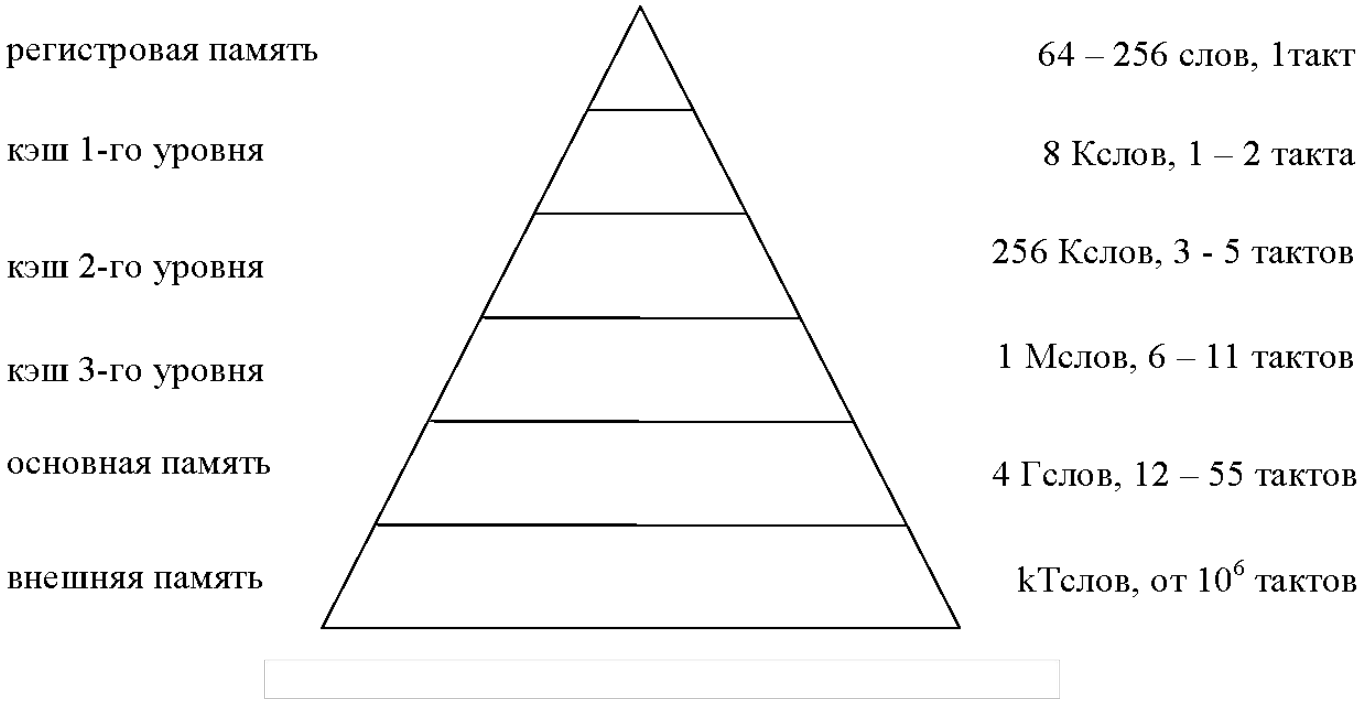
Идея **иерархической (многоуровневой)** организации памяти заключается в использовании на одном компьютере нескольких уровней памяти, которые характеризуются разным временем доступа к памяти и объемом памяти. (Время доступа к памяти это время между операциями чтения/записи, которые выполняются по случайным адресам.) Основой для иерархической организации памяти служит принцип **локальности ссылок во времени и в пространстве.**

- **Локальность во времени** состоит в том, что процессор многократно использует одни и те же команды и данные.
- **Локальность в пространстве** состоит в том, что если программе нужен доступ к слову с адресом А, то скорее всего, следующие ссылки будут к адресам, расположенным по близости с адресом А.

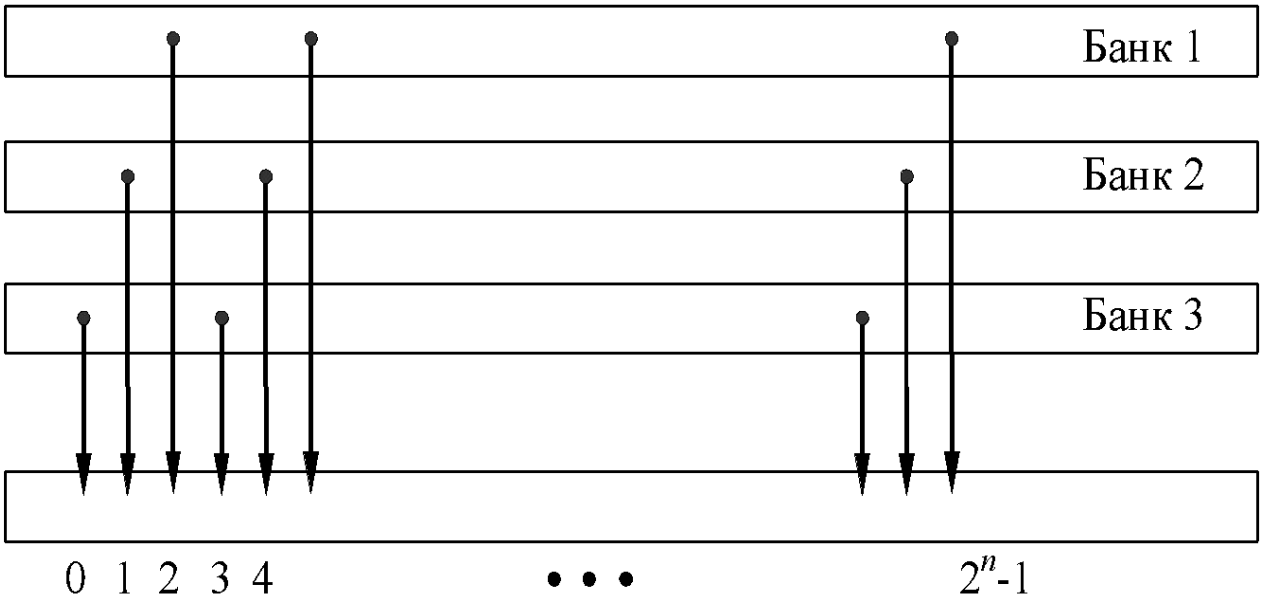
Из свойства локальности ссылок следует, что в типичном вычислении обращения к памяти концентрируются вокруг небольшой области адресного пространства и более того, выборка идет по последовательным адресам. Время доступа к иерархически организованной памяти уменьшается благодаря следующему

- **сокращению количества обращений к оперативной памяти**
- **совмещению обработки текущего фрагмента программы и пересылки данных из основной памяти в буферную память.**

Схема иерархического построения памяти



Интерливинг



Адресное пространство

Организация кэш-памяти

Кэш-память это высокоскоростная память небольшего размера с прямым доступом. Она предназначена для временного хранения фрагментов кода и данных. Кэш-память охватывает все адресное пространство памяти, но в отличие от оперативной памяти, она не адресуема и невидима для программиста.

Схема построения кэш-памяти

- Кэш-память построена на принципе локальности ссылок во времени и в пространстве.
- Кэш-контроллер загружает копии программного кода и данных из ОП в кэш-память блоками, равными размеру строки за один цикл чтения. Процессор читает из кэш-памяти по словам.
- Кэш-контроллер перехватывает запросы процессора к основной памяти и проверяет, есть ли действительная копия информации в кэш-памяти.

Структура кэш-памяти

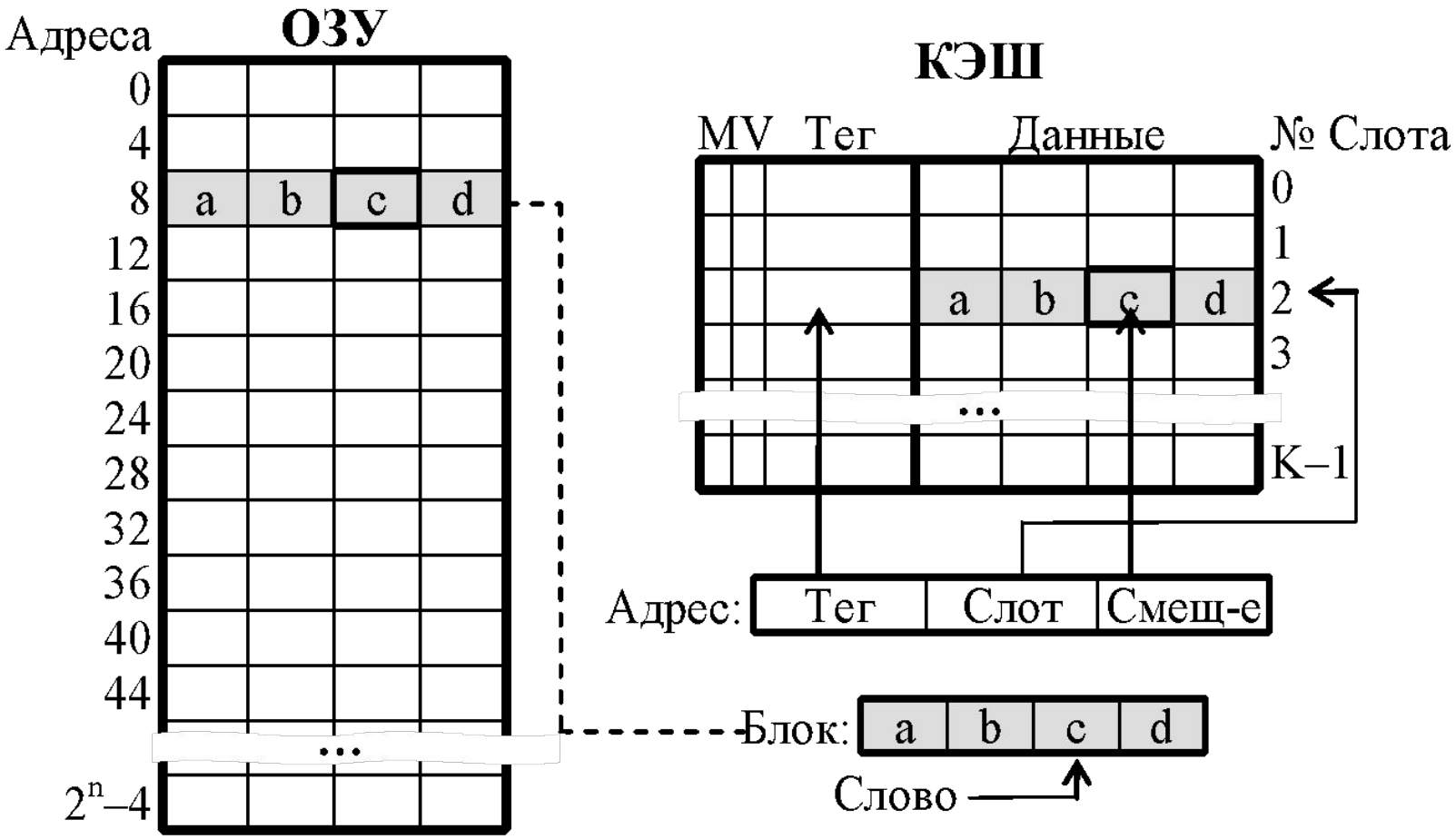
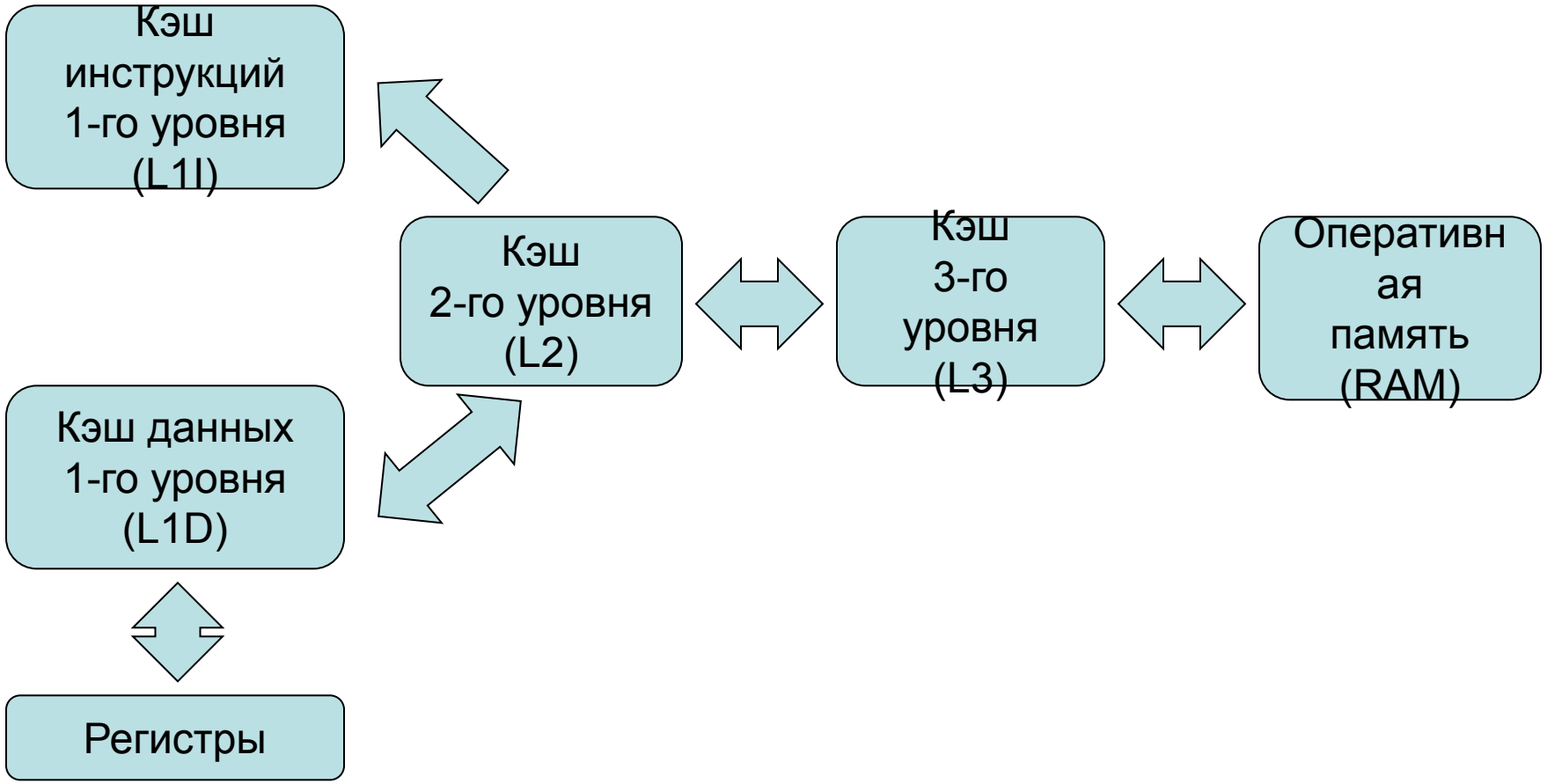


Схема иерархического построения памяти



Организация кэш-памяти

Когда контроллер помещает данные в кэш- память?

- Загрузка по требованию **(on demand).**
- Спекулятивная загрузка **(speculative load).**
  Алгоритм предполагает помещать данные в кэш-память задолго до того, как к ним произойдет реальное обращение. У кэш-контроллера есть несколько алгоритмов, которые указывают, какие ячейки памяти потребуются процессору в ближайшее

  время.

Организация кэш-памяти

Когда контроллер выполняет поиск данных в памяти?

- после фиксации промаха (*сквозной просмотр*).
- одновременно с поиском блока в кэш-памяти, в случае кэш-попадания, обращение к оперативной памяти прерывается (*отложенный просмотр*).

Основные вопросы организации кэш-памяти

- Алгоритм отображения адресов основной памяти в кэш-память.
- Алгоритм записи данных и команд из кэш-памяти в основную память.
- Алгоритм замещения строки в кэш-памяти.
- Размер кэш-памяти.
- Длина строки в кэш-памяти.

Алгоритмы отображения

- Прямой **(direct mapping).**
- Ассоциативный **(full associative mapping).**
- Множественно-ассоциативный **(set-associative mapping).**

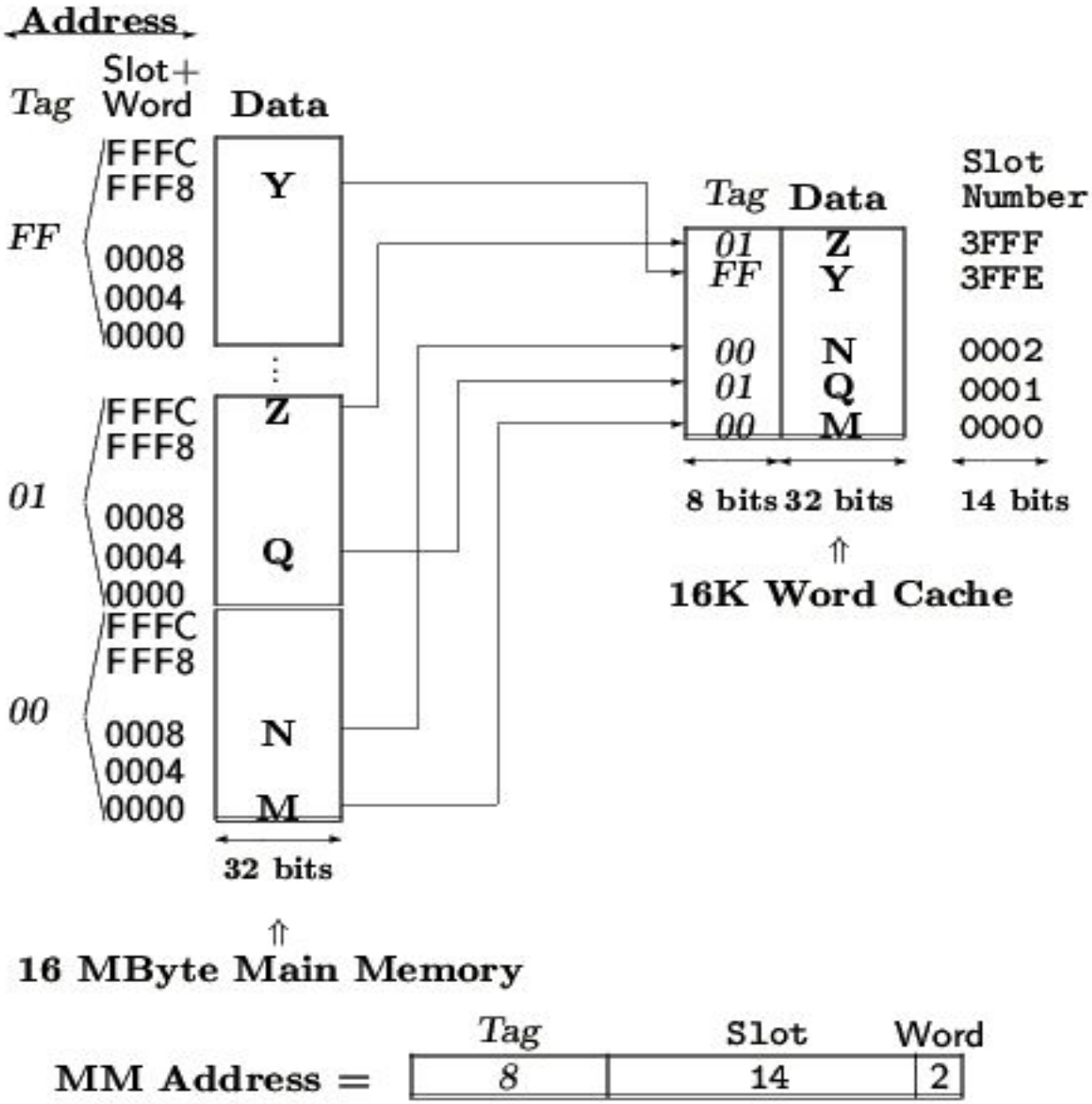


Figure 2: **Direct Mapping**

Пример «буксования» кэш-памяти **(32 K)** (***cache trashing)***

```
real*8 a(4096),b(4096),c(4096)
    common a,b,c
    do i=1,4096
      c(i)=a(i)+b(i)
    enddo
```

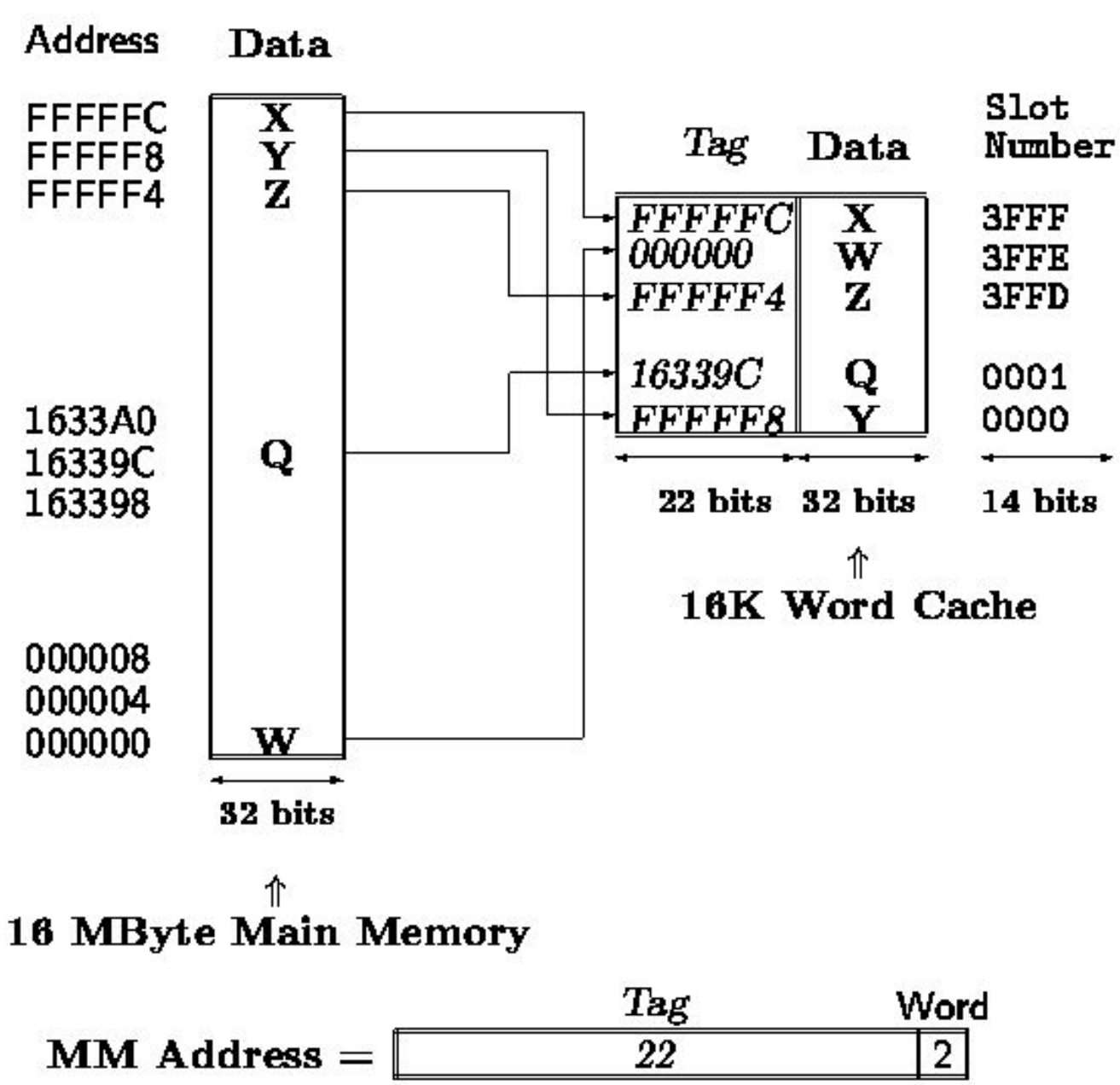
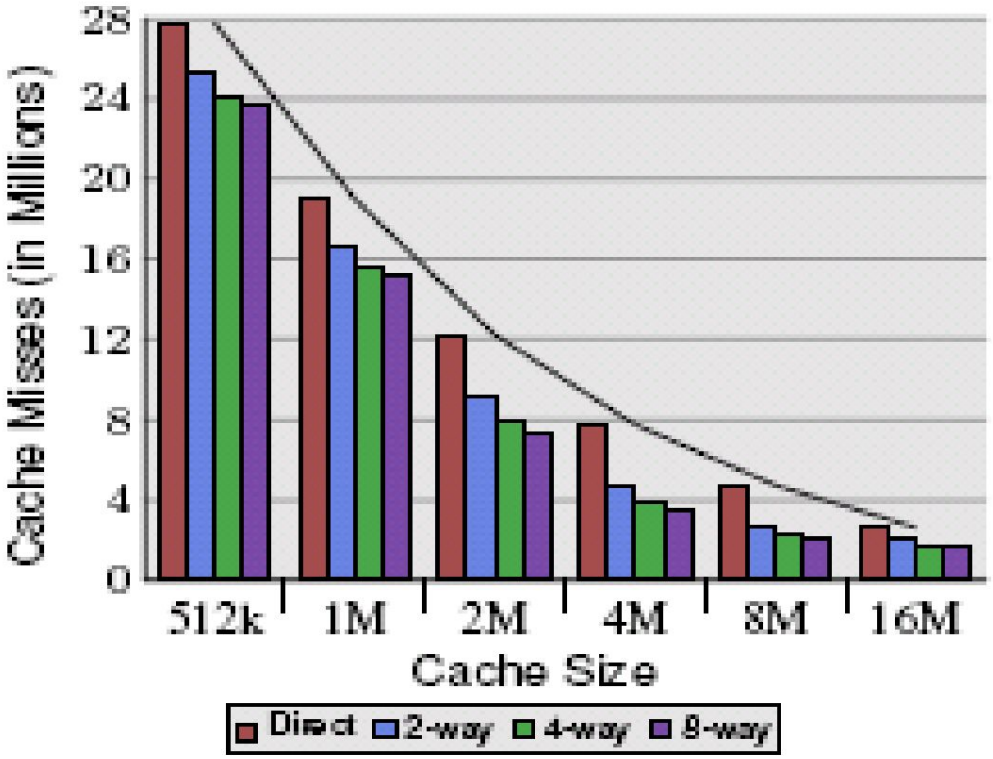
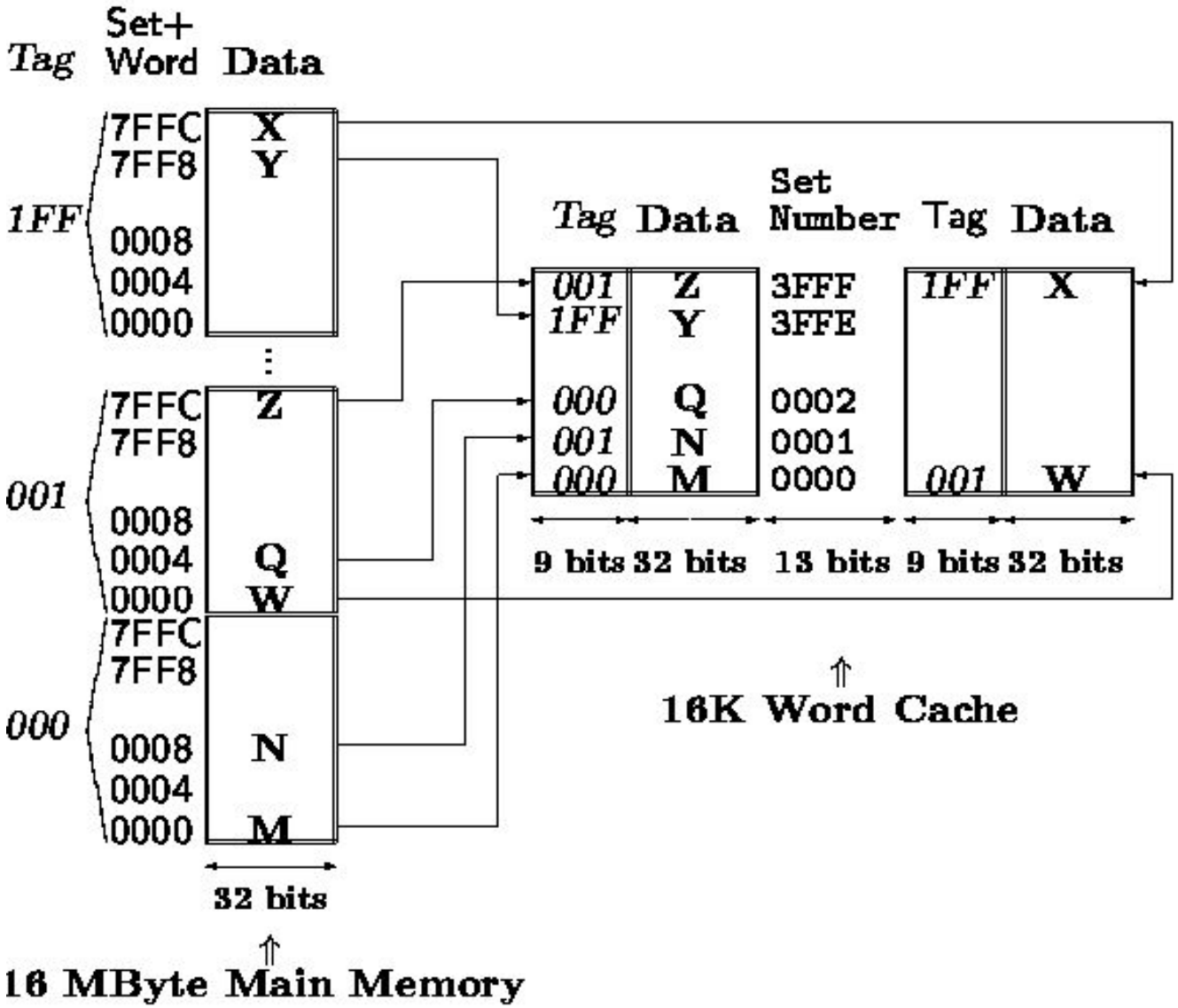


Figure 3: **Associative Mapping**

Зависимость количества промахов в кэш-память в зависимости от объема кэш-памяти и степени ассоциативности для длины троки 32 байта





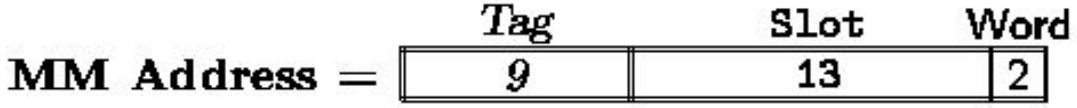


Figure 4: **Two-way Set-Associative Mapping**

Сравнение алгоритмов отображения адресов

- Прямой
  - 1 блок – 1 строка
  - Плюс: быстрый поиск, маленькие теги, простая реализация
  - Минус: пробуксовка кэша
- (Полностью) ассоциативный
  - 1 блок – любая строка
  - Плюс: нет пробуксовки кэша
  - Минус: медленный поиск, большие теги, сложная реализация
- Множественно-ассоциативный
  - 1 блок – несколько строк
  - Компромиссный вариант

Алгоритмы записи

- Сквозная запись **(Write Through (WT)).**
- Сквозная запись с буфери-зацией **(Write Combining).**
- Обратная запись **(Write Back (WB)).**

Алгоритмы замещения кэш-строк

- **Least Recently Used (LRU)**
- **Most Recently Used (MRU)**
- **Pseudo-Least Recently Used (PLRU)**

Алгоритм замещения (алгоритм псевдо-**LRU**)

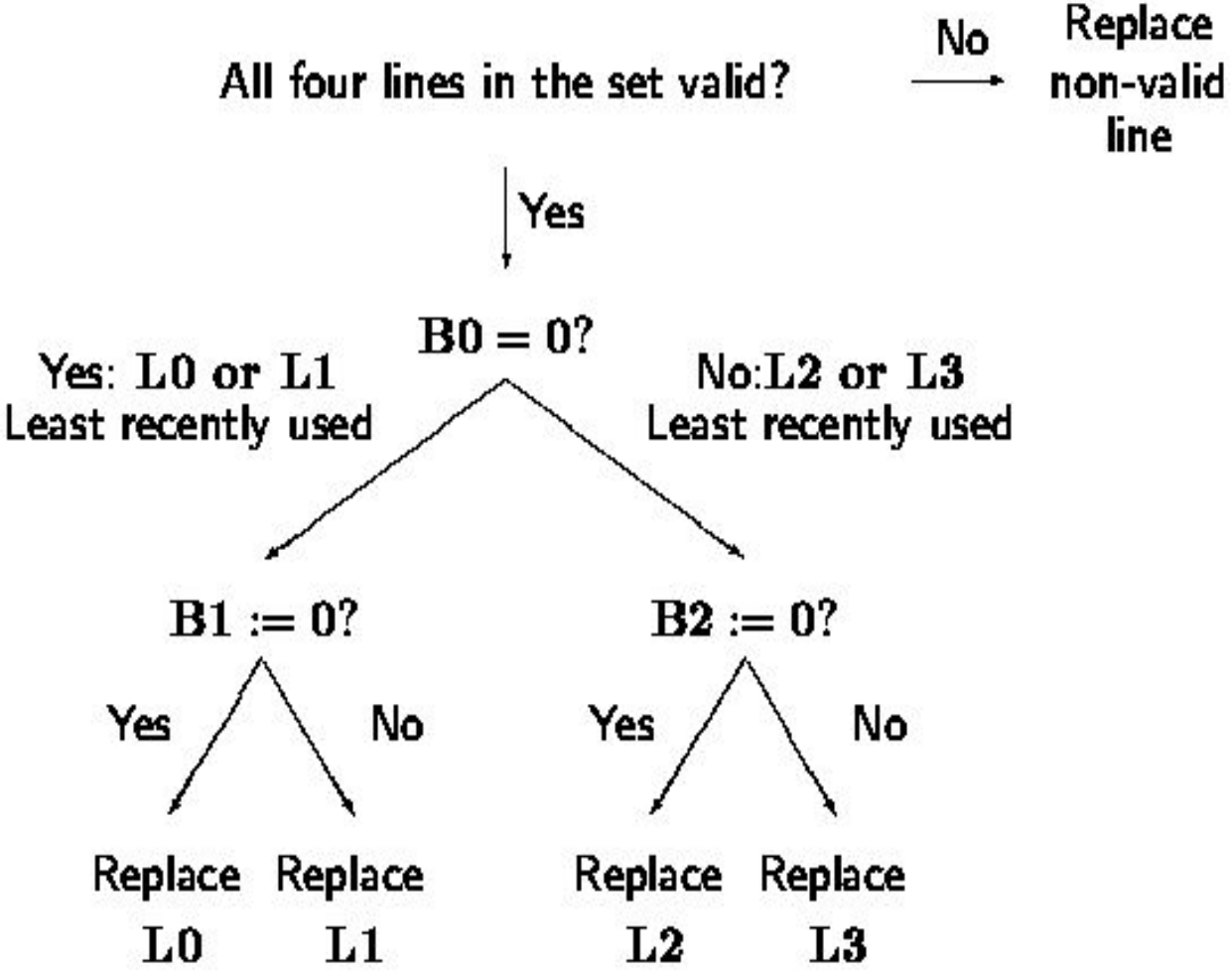


Figure 6: **Intel 80486 on-chip cache replacement algorithm**

Каким должен быть размер линии кэш-памяти?

- Размер линии должен быть как минимум в ширину канала памяти

- **Большой размер**

  -- Более эффективное использование канала памяти при последовательном доступе

  -- Позволяет уменьшать **«**ассоциативность**»** кэша и количество линий

- **Маленький размер**

  -- Более эффективное использование канала памяти при произвольном доступе

  -- Заполнение можно делать за одну транзакцию к памяти

Какими должны быть основные параметры кэша?

- **Размер кэша**
  - **Большой**, чтобы вместить рабочие данные
  - **Маленький**, для быстрого доступа
- **Степень ассоциативности кэша**
  - **Большая**, чтобы избегать пробуксовки
  - **Маленькая**, для быстрого доступа
- **Размер строки кэша**
  - **Большой**, чтобы использовать локальность
  - **Большой**, чтобы уменьшить теги
  - **Маленький** (доля полезных данных в кэше больше, если данные в памяти распределены произвольным образом)

Процессор	Pentium 4, Xeon (Northwood)	Itanium2	Opteron	Alpha 21264	PowerPC 970FX
Программные регистры	8 целочисл. (32 бит), 8 веществ. (80 бит), 16 векторных (128 бит)	128 целочисл. (64 бит), 128 веществ. (82 бит), 64 предикатных (1 бит), 8 регистров ветвлений (64 бит), 128 прикладных регистра	16 целочисл. (64 бит), 8 веществ. (80-бит), 16 векторных (128-бит)	32 целочисл. (64 бит), 32 веществ. (64 бит)	32 целочисл (64 бит), 32 веществ. (64 бит), 16 векторных (128-бит)
Аппаратные регистры	целочисл. 128 (32 бит), веществ. 128 (128 бит)	соответствуют программным	40 целочисл., 120 веществ.	80 целочисл., 72 веществ.	32 + 48 целочисл., 32 + 48 веществ., 16 + 16 векторных
Кэш данных L1	8 KB, 4-way, строка 64 B,	16 KB, 4-way, строка 64 B,	64 KB, 2-way, строка 64 B,	64 KB, 2-way, строка 64 B,	32 KB, 2-way, строка 128 B,
Кэш команд L1	кэш трасс, 12 K микроопераций, 8-way	16 KB, 4-way, строка 64 B,	64 KB, 2-way, строка 64 B	64 KB, 2-way, строка 64 B,	64 KB, прямого отображения,
Кэш L2	512 KB, 8-way, строка 64B,	256 KB, 8-way, строка 128B,	1 MB, 16-way, строка 64 B,	1 – 4 MB, внешний,	512 KB, 8-way, строка 128B,
Кэш L3		1.5 – 9 MB, 12-way, строка 128B,			

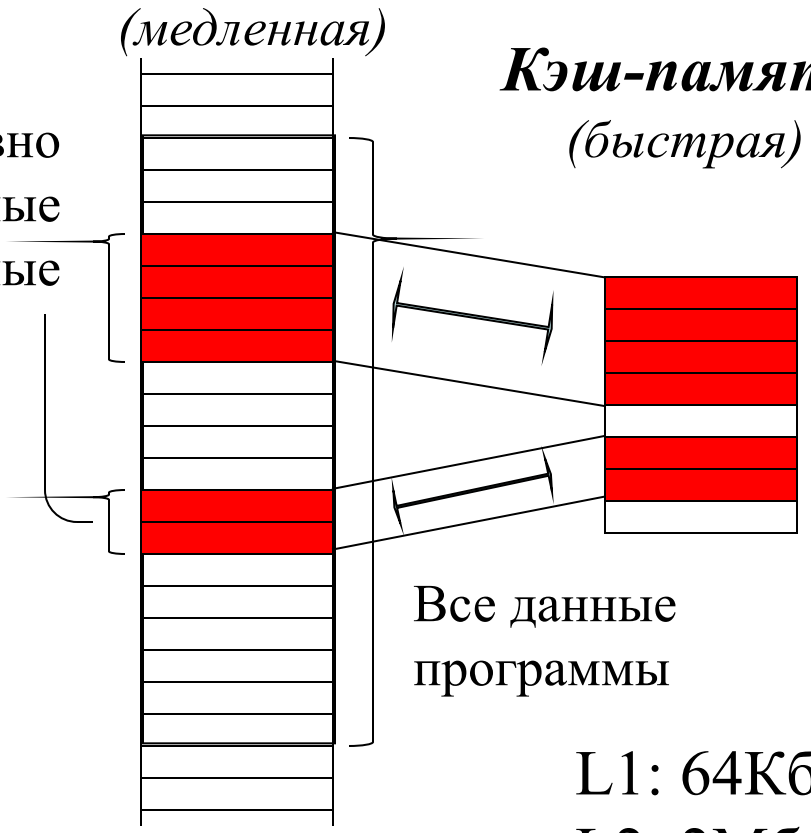
Эффективное использование иерархии памяти

- **Объем обрабатываемых данных**
- **Обход данных**

Схема иерархической памяти

Программа:

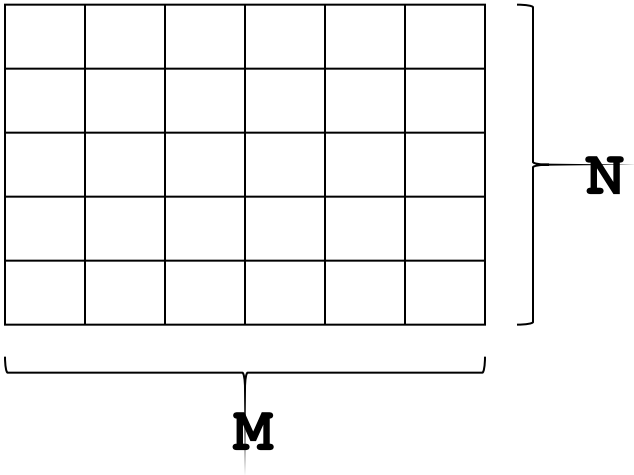
```
int prog()
{  ...
   ...
   ...
   ...
   for (i…)
    for (j…)
    {a[i][j]…
     b[j]…
    }
   ...
   ...
}
```



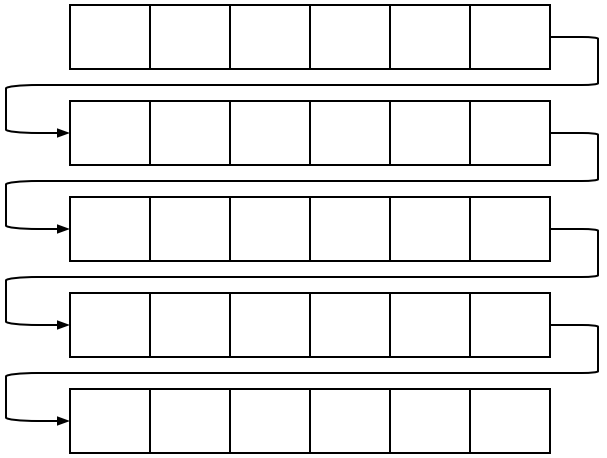
Последовательный обход данных (Си)

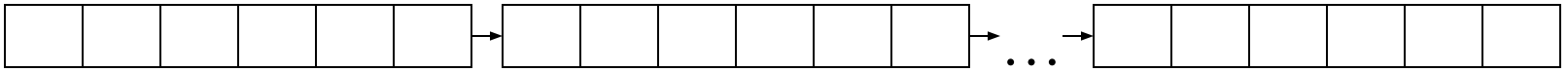
2D массив:

```
float X[N][M];
```



Размещение массива в памяти:





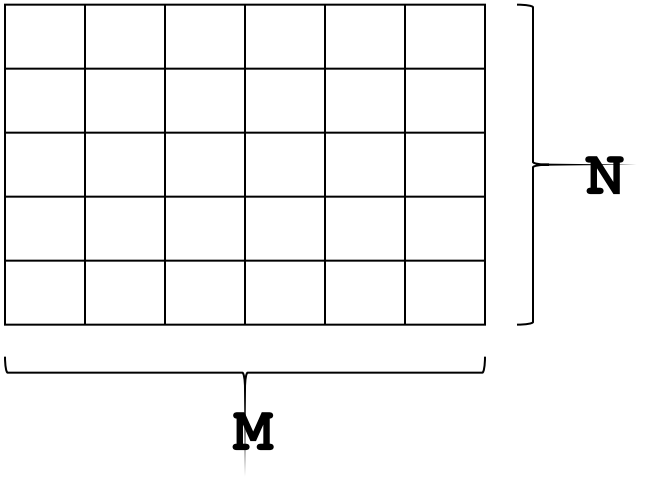
Строка 0 Строка 1 Строка N-1

```
for (i=0;i<N;i++)          // цикл по строкам
  for (j=0;j<M;j++)        // цикл по столбцам
    X[i][j]=expr(i,j);
```

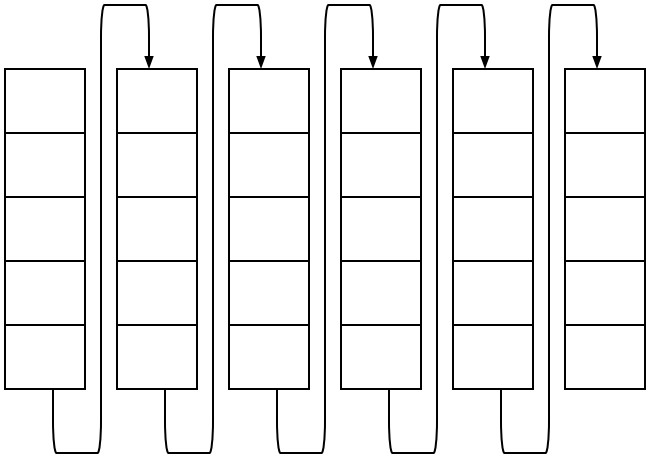
Последовательный обход данных (Фортран)

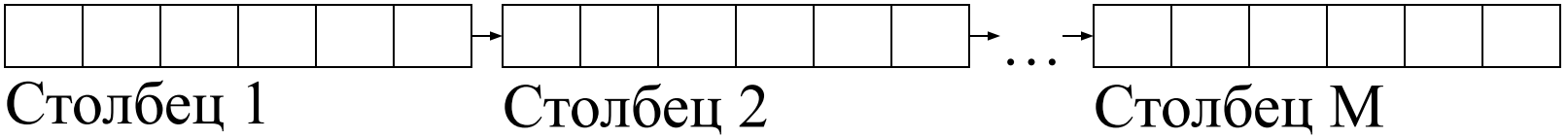
2D массив:

```
real X(N,M)
```



Размещение массива в памяти:





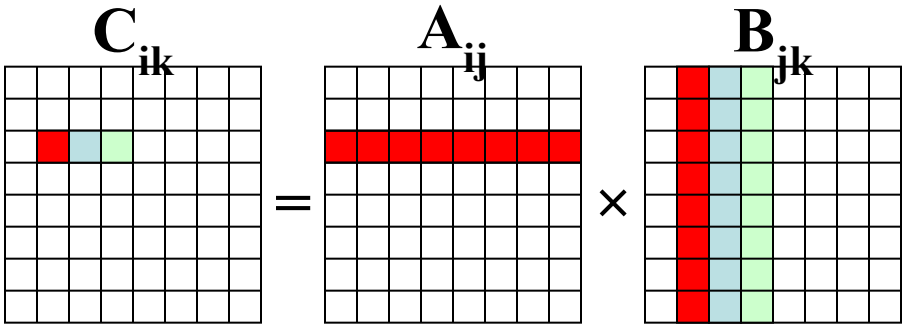

```
      do 10 j=1,M              цикл по столбцам
        do 10 i=1,N            цикл по строкам
10        X(i,j)=expr(i,j)
```

Пример обхода данных

Перемножение матриц: C[Nx][Nz]=A[Nx][Ny]*B[Ny][Nz]

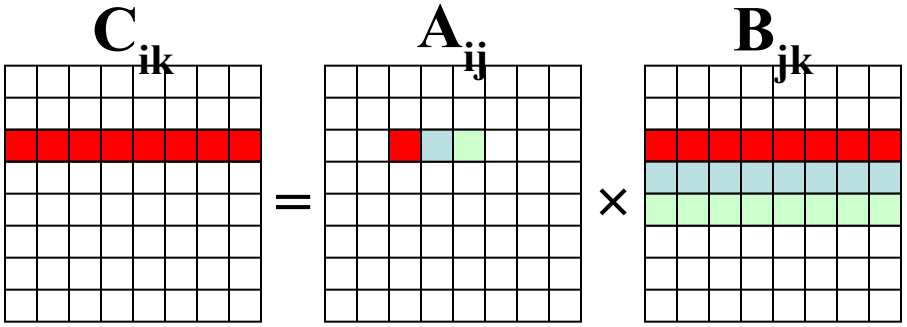
$$C_{ik} = \sum_{j=0}^{Ny-1} A_{ij} B_{jk}$$

Медленно:




```
for(i=0;i<Nx-1;i++)
 for(k=0;k<Nz-1;k++)
  for(j=0;j<Ny-1;j++)
   C[i][k]+=A[i][j]*B[j][k];
```

Быстрее:




```
for(i=0;i<Nx-1;i++)
 for(j=0;j<Ny-1;j++)
  for(k=0;k<Nz-1;k++)
   C[i][k]+=A[i][j]*B[j][k];
```

Результаты теста: перемножение матриц

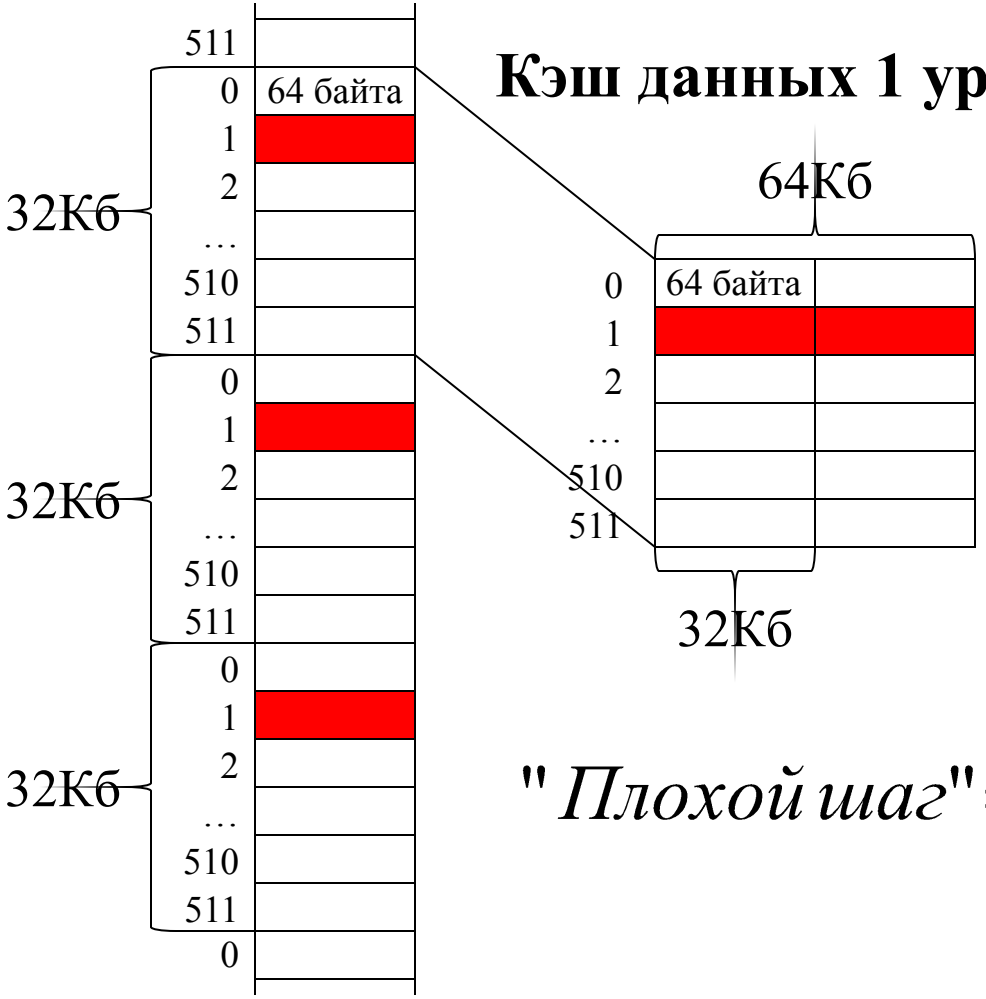
Размеры матриц
800*800

Схема программы:

```
for (…)
  for (…)
    for (…)
      C[i][k]+=A[i][j]*B[j][k];
```

Эффект «буксования» кэша

Оперативная память



Кэш данных 1 уровня процессора Alpha 21264

Параметры:

- Объем: 64 Кб
- 2-канальный множественно-ассоциативный

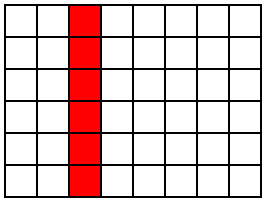
$$"Плохой\ шаг" = \frac{64Кб}{2} = \mathbf{32\ Кб}$$

и все кратные

32 Кб = 8192 float (real*4) = 4096 double (real*8)

Пример «буксования» кэша: 2D массив

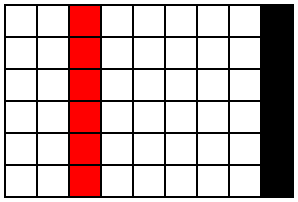
`X[Ny][Nx];`



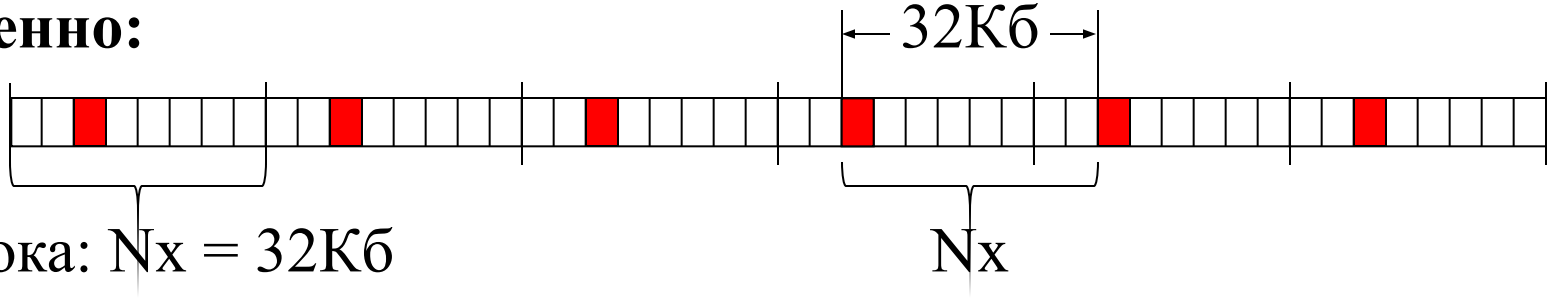
Выполняем обход по столбцам

в программе на языке Си

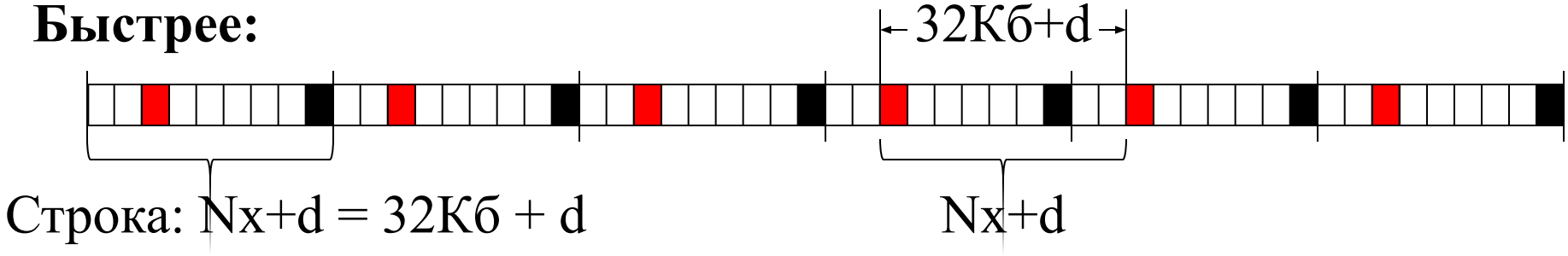
`X[Ny][Nx+d];`



Медленно:



Быстрее:

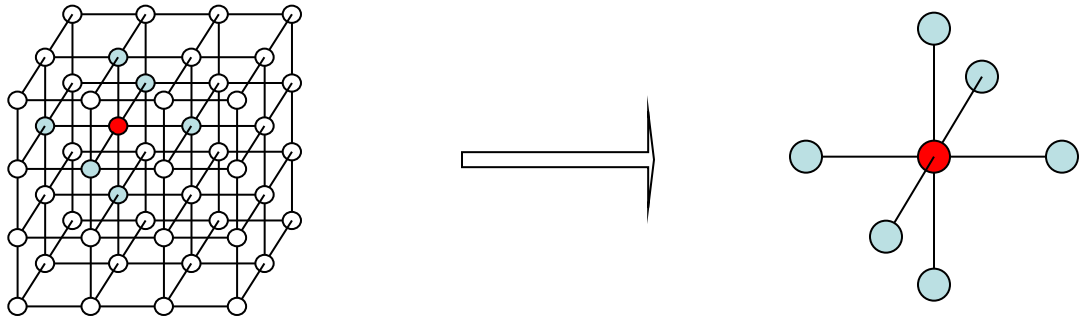


d ≥ размер строки кэш-памяти (для Alpha 21264 – 64 байта)

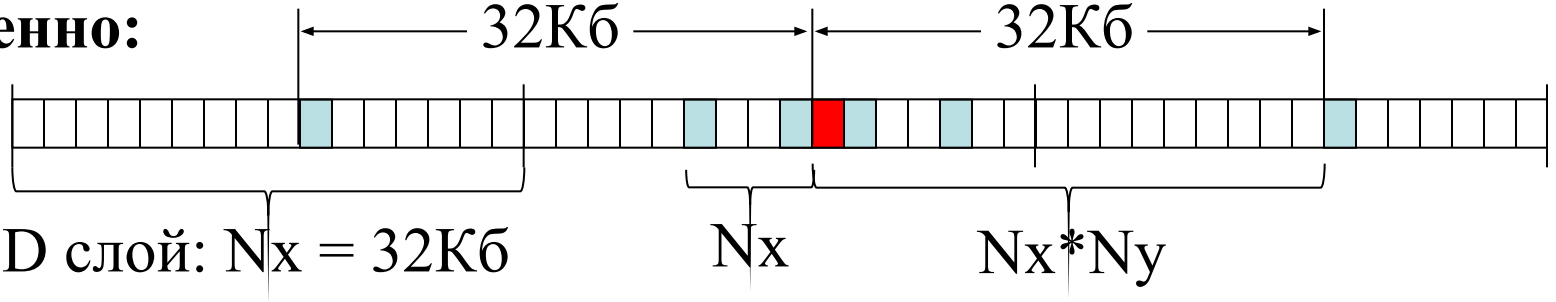
Пример «буксования» кэша: 3D массив

`X[Nz][Ny][Nx];`

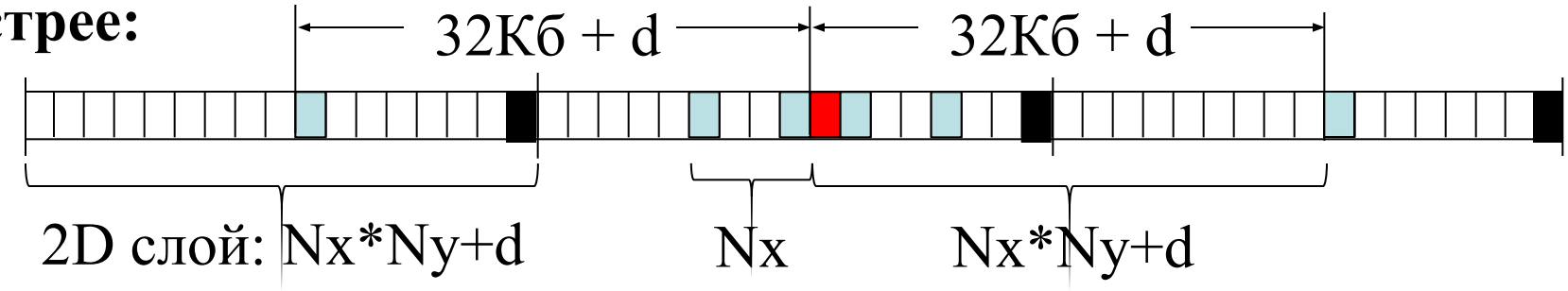
Семиточечная схема:



Медленно:



Быстрее:



d ≥ размер строки кэш-памяти (для Alpha 21264 – 64 байта)

Результаты теста: решение уравнения Пуассона в трехмерной области методом Якоби

Размер массива: N × N × N

Размер 2D слоя для N=128: 128*128 = 16384 элементов = **64 Кб**

Рекомендации

- Объем активно используемых данных не должен превосходить размер кэша
- По возможности используйте последовательный обход данных
- Избегайте одновременного использования данных, расположенных в памяти на расстоянии 32Кб (и кратном 32Кб)

Компиляторы языка Си на МВС-1000

- Compaq C Compiler v6.4.9
  команда: **ccc**
    оптимизирован для процессоров Alpha
- GNU C Compiler v2.91
  команда: **gcc** или **cc**
    формирует код для любого процессора
    старая версия компилятора, формирует очень медленный код

Для компиляции Ваших программ используйте Compaq C Compiler (команда: **ccc**)

Замечание:

Команда **mpicc** использует Compaq C Compiler

Ключи оптимизации компилятора ccc

–O0 Отключает оптимизацию.

–O1 Локальная оптимизация и распознавание общих подвыражений. Начальная глобальная оптимизация.

Используется по умолчанию.

–O

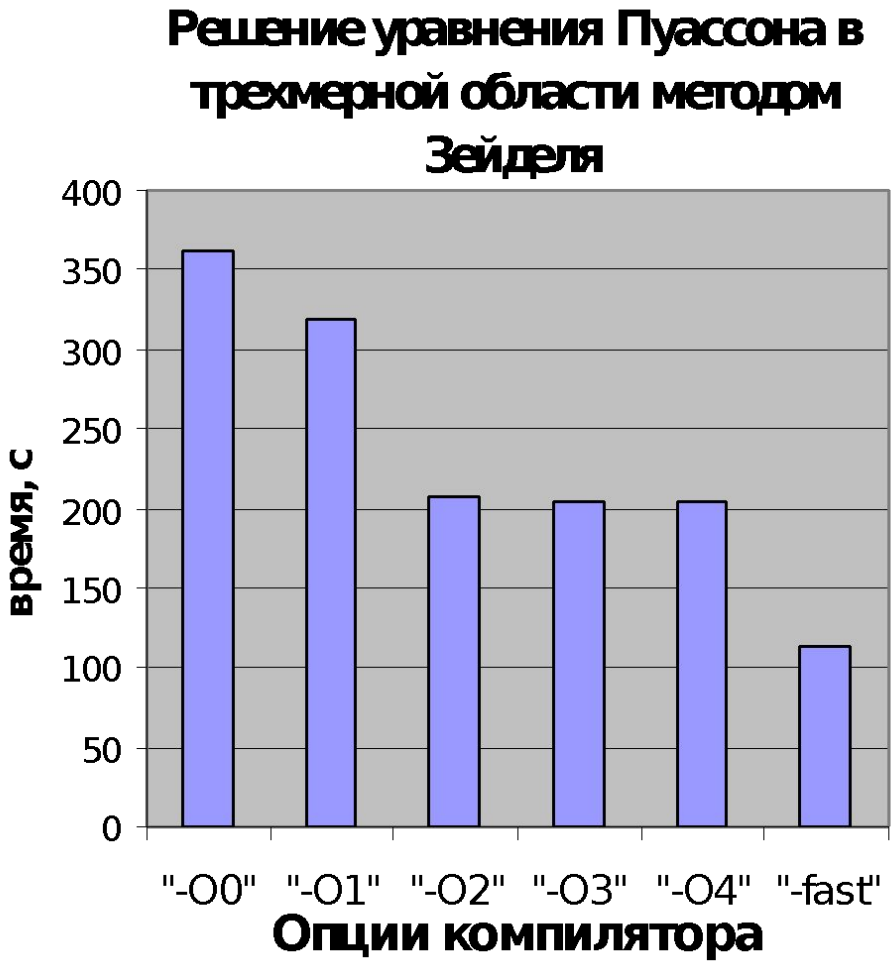
–O2 Встраивание небольших статических подпрограмм. Глобальная оптимизация, приводящая к увеличению размера кода: раскрытие умножения и деления (используя сдвиги), раскрутка циклов, дублирование кода для избежания ветвлений.

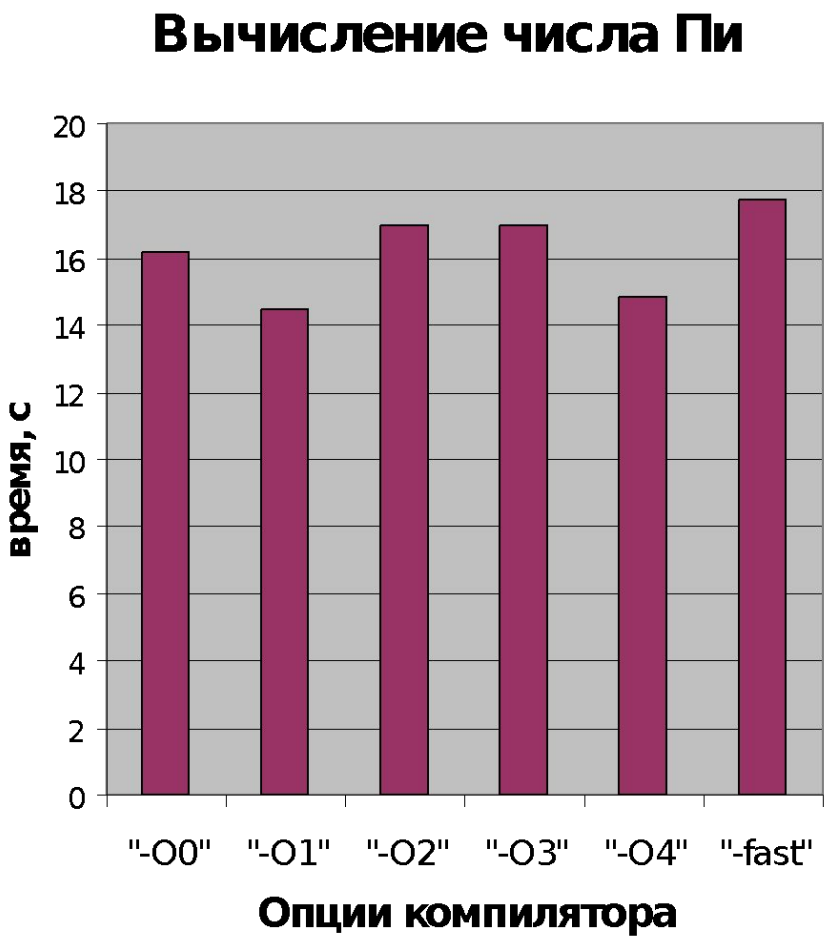
–O3 Встраивание небольших глобальных подпрограмм.

–O4 Программная конвейеризация.

–fast -O3 плюс дополнительная оптимизация, включая оптимизацию вещественных вычислений, приводящую к некоторой потере точности.

Примеры использования различных ключей оптимизации





Помощь компилятору

1. Используйте средства языка для более точного описания природы и способа использования объектов языка
   - Старайтесь не использовать глобальных переменных
   - Используйте различные переменные для разных вычислений
2. Компилятору должно быть ясно, какие зависимости имеются между данными в программе
   - Старайтесь не использовать указатели
   - Компилируйте все модули программы сразу
3. Не стоит делать вручную то, что может сделать компилятор
   - Не разворачивайте циклы
   - Пишите более ясный и простой код, особенно внутри

Точность вычислений

- Определяется:
  - типом данных (float/real*4, double/real*8, real*16)
  - размеров вещественных регистров процессора (64 бита для процессоров Alpha)
- Не зависит от компилятора (ссс, forte)
- Может меняться в результате преобразования программы:
  - изменение порядка данных
  - изменение порядка операций
  - эквивалентные преобразования выражений
  - оптимизация с помощью компилятора