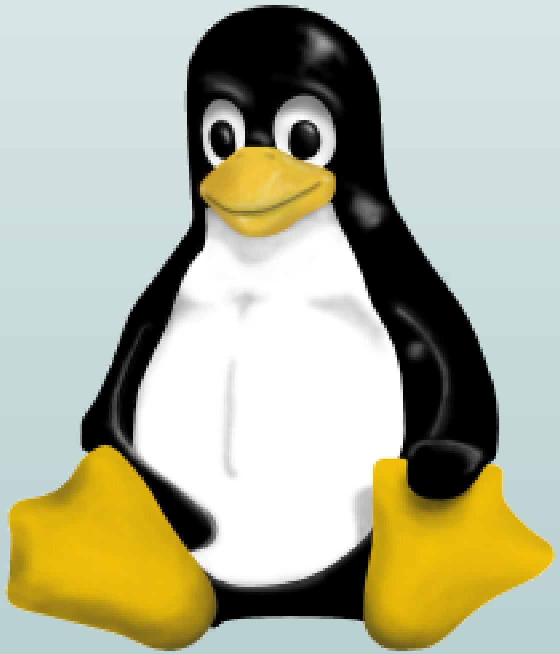
Linux Kernel Hacking Free Course, 3rd edition

M. Cesati
University of Rome “Tor Vergata”

Linux, the caches, and you



March 8, 2006

About this lecture

While preparing this lecture, I asked myself:

> “*What have you learned from Linux that might be of interest for any programmer?*”

About this lecture

While preparing this lecture, I asked myself:

> "*What have you learned from Linux that might be of interest for any programmer?*"

I was quickly forced to focus on some specific topic, otherwise this lecture would become way too long

And the winner is: **Linux and the caches (and you, of course!)**

Roadmap (sort of)

So, now I’ll try my best to:

- explain the “cache problem”

Roadmap (sort of)

So, now I'll try my best to:

- explain the "cache problem"

- present some scenarios in which caches make really a difference

Roadmap (sort of)

So, now I'll try my best to:

- explain the "cache problem"
- present some scenarios in which caches make really a difference
- introduce some techniques (learned from the Linux source code) to fully exploit the caches

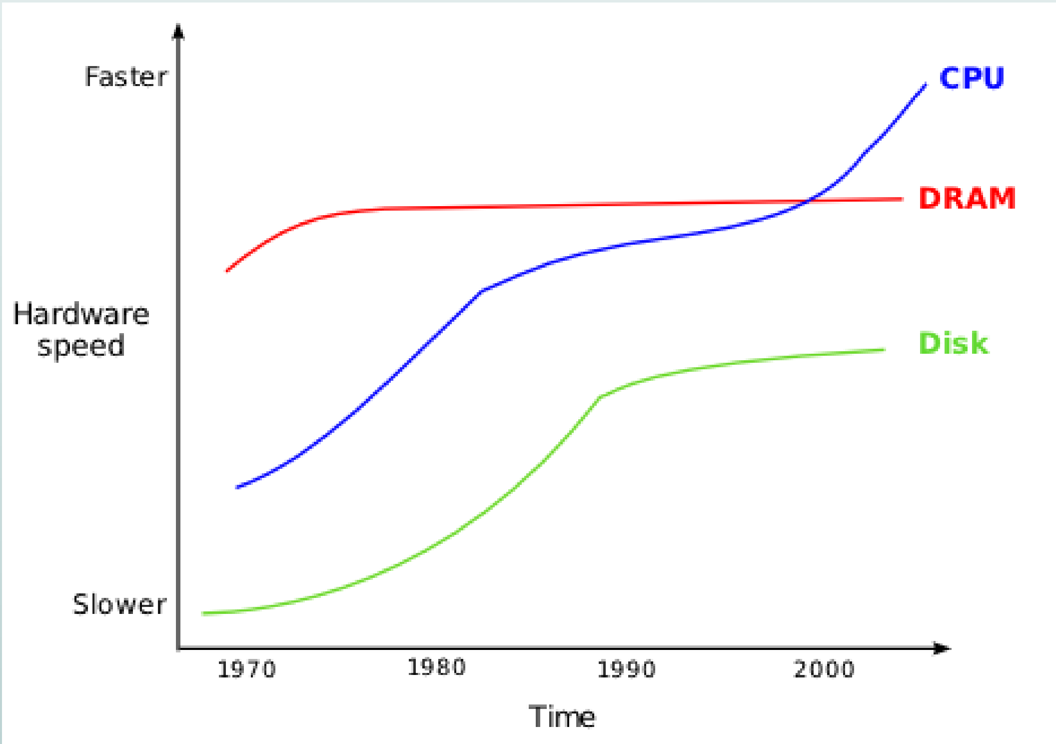
Roadmap (sort of)

So, now I'll try my best to:

- explain the "cache problem"
- present some scenarios in which caches make really a difference
- introduce some techniques (learned from the Linux source code) to fully exploit the caches
- keep you awake!

Memory can't keep up

CPU speed grows much faster than hard disk and DRAM speed:



What's the matter, then?

It is a matter of time:

- Typical random access time for a PC100 (S)DRAM chip is 40 nanoseconds
- A 2 GHz CPU may execute up to 80 register operations in 40 nanoseconds

What's the matter, then?

It is a matter of time:

- Typical random access time for a PC100 (S)DRAM chip is 40 nanoseconds
- A 2 GHz CPU may execute up to 80 register operations in 40 nanoseconds

and also a matter of costs:

- DRAM cells are relatively cheap: $\approx$ 0.075 €/ MB
- SRAM cells (registers) are costly (they require six transistors per bit instead of just one, more wiring, more space): $\approx$ 0.75 €/ MB

The solution: caching, caching and more caching

Modern computer architectures use a large amount of slow and cheap DRAM cells, as well as a small amount of fast and costly SRAM cells:

- L3 (unified) cache
- L2 (unified) cache
- L1 unified cache / L1 data cache
- L1 instruction cache / Trace cache

The solution: caching, caching and more caching

Modern computer architectures use a large amount of slow and cheap DRAM cells, as well as a small amount of fast and costly SRAM cells:

- L3 (unified) cache
- L2 (unified) cache
- L1 unified cache / L1 data cache
- L1 instruction cache / Trace cache
- Translation Lookaside Buffer (TLB)

The solution: caching, caching and more caching

Modern computer architectures use a large amount of slow and cheap DRAM cells, as well as a small amount of fast and costly SRAM cells:

- L3 (unified) cache
- L2 (unified) cache
- L1 unified cache / L1 data cache
- L1 instruction cache / Trace cache
- Translation Lookaside Buffer (TLB)
- Branch Target Instruction Cache

The solution: caching, caching and more caching

Modern computer architectures use a large amount of slow and cheap DRAM cells, as well as a small amount of fast and costly SRAM cells:

- L3 (unified) cache
- L2 (unified) cache
- L1 unified cache / L1 data cache
- L1 instruction cache / Trace cache
- Translation Lookaside Buffer (TLB)
- Branch Target Instruction Cache
- Write combining registers

The solution: caching, caching and more caching

Modern computer architectures use a large amount of slow and cheap DRAM cells, as well as a small amount of fast and costly SRAM cells:

- L3 (unified) cache
- L2 (unified) cache
- L1 unified cache / L1 data cache
- L1 instruction cache / Trace cache
- Translation Lookaside Buffer (TLB)
- Branch Target Instruction Cache
- Write combining registers
- CPU general-purpose registers

A new idea?! (A digression)

> "Ideally one would desire an indefinitely large memory capacity such that any particular [word] would be immediately available. . . It does not seem possible to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A new idea?! (A digression)

> “Ideally one would desire an indefinitely large memory capacity such that any particular [word] would be immediately available... It does not seem possible to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.”

Preliminary discussion of the logical design of an electronic computing instrument, Burks, Goldstine, von Neumann, 1946

Locality principles

Why is caching beneficial?

Locality principles

Why is caching beneficial?

Spatial locality: In a short time frame, the executed instructions

- are generally in a small area of memory
- access memory cells that are close to one another

Locality principles

Why is caching beneficial?

Spatial locality: In a short time frame, the executed instructions

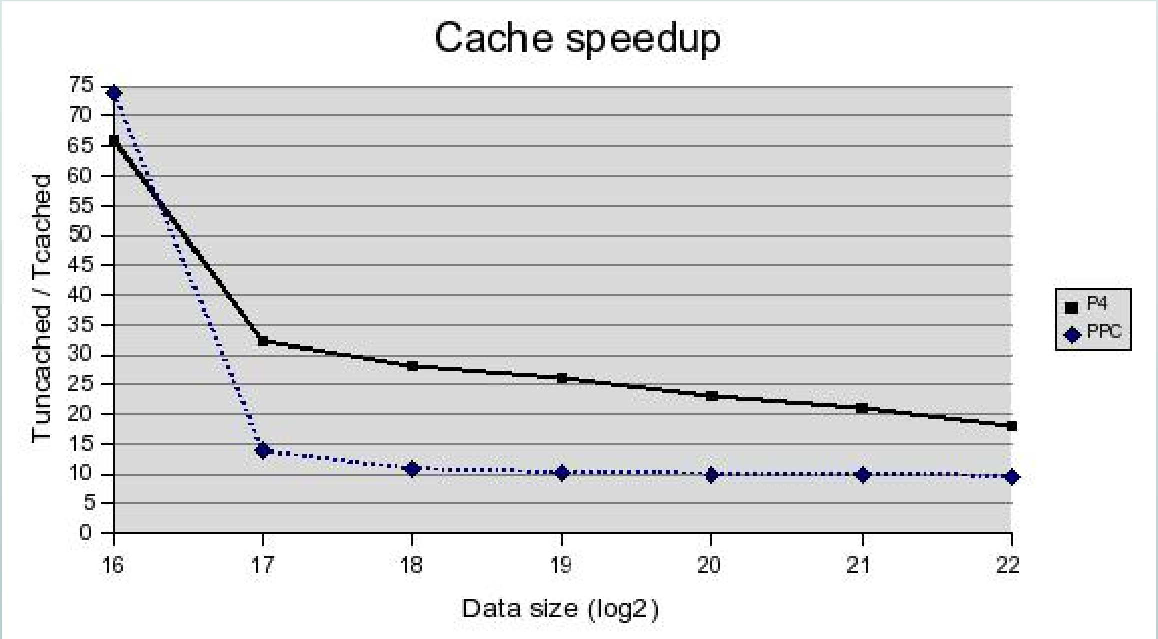
- are generally in a small area of memory
- access memory cells that are close to one another

Temporal locality: In a short time frame, the executed instructions

- are likely to be executed again
- access memory cells that will likely be accessed again

Performance gain of caches

We measured execution times of a discrete FFT on increasingly larger data sets with and without L1/L2 caches, both on an Intel Pentium 4 and on a Freescale MPC7447A PowerPC



The bottom rule

When coding, you should always keep in mind the cache problem:

The bottom rule

When coding, you should always keep in mind the cache problem:

- How is your code accessing data? Is it fully exploiting the caches?

The bottom rule

When coding, you should always keep in mind the cache problem:

- How is your code accessing data? Is it fully exploiting the caches?
- What are the most frequently executed functions/loops/fragments in your code? Are they optimized for fully exploiting the caches?

The bottom rule

When coding, you should always keep in mind the cache problem:

- How is your code accessing data? Is it fully exploiting the caches?
- What are the most frequently executed functions/loops/fragments in your code? Are they optimized for fully exploiting the caches?

In many cases, *cache-conscious* algorithms achieve significant performance gains

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:

read data from array `x[0..n-1]`

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:

- read data from array `x[0..n-1]`
- skip `d` elements of `x`

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:
- read data from array `x[0..n-1]`
- skip `d` elements of `x`
- in case of overflow:

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:
- read data from array `x[0..n-1]`
- skip `d` elements of `x`
- in case of overflow:
  - wrap around the index `j`

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:
- read data from array `x[0..n-1]`
- skip `d` elements of `x`
- in case of overflow:
  - wrap around the index `j`

The benchmark consists of measuring the execution time of this loop for many different values of array size `n` and skipping delta `d`, and dividing the time by `reps`

A “real-world” code fragment as a cache benchmark

```
for (i=0; i<reps; ++i) {
    t += x[j];
    j += d;
    if (j >= n)
        j -= n;
}
```

in `reps` iterations:
- read data from array `x[0..n-1]`
- skip `d` elements of `x`
- in case of overflow:
  - wrap around the index `j`

The benchmark consists of measuring the execution time of this loop for many different values of array size `n` and skipping delta `d`, and dividing the time by `reps`

The number of iterations `reps` is large (e.g., 10000000), so each benchmark value is roughly the average execution time of one access to the array `x`

Measuring the execution time

In order to measure the execution time of a code fragment, we can use the Time Stamp Counter device found in many CPUs (yet another fine trick learned from the Linux source code!)

Measuring the execution time

In order to measure the execution time of a code fragment, we can use the Time Stamp Counter device found in many CPUs (yet another fine trick learned from the Linux source code!)

For example, in the IA-32 microprocessors, a 64-bit register stores a counter that is incremented once every CPU clock signal

Measuring the execution time

In order to measure the execution time of a code fragment, we can use the Time Stamp Counter device found in many CPUs (yet another fine trick learned from the Linux source code!)

For example, in the IA-32 microprocessors, a 64-bit register stores a counter that is incremented once every CPU clock signal

The `rdtsc` Assembly language instruction reads the value in this register and stores it into the `eax` and `edx` general-purpose registers

Measuring the execution time

In order to measure the execution time of a code fragment, we can use the Time Stamp Counter device found in many CPUs (yet another fine trick learned from the Linux source code!)

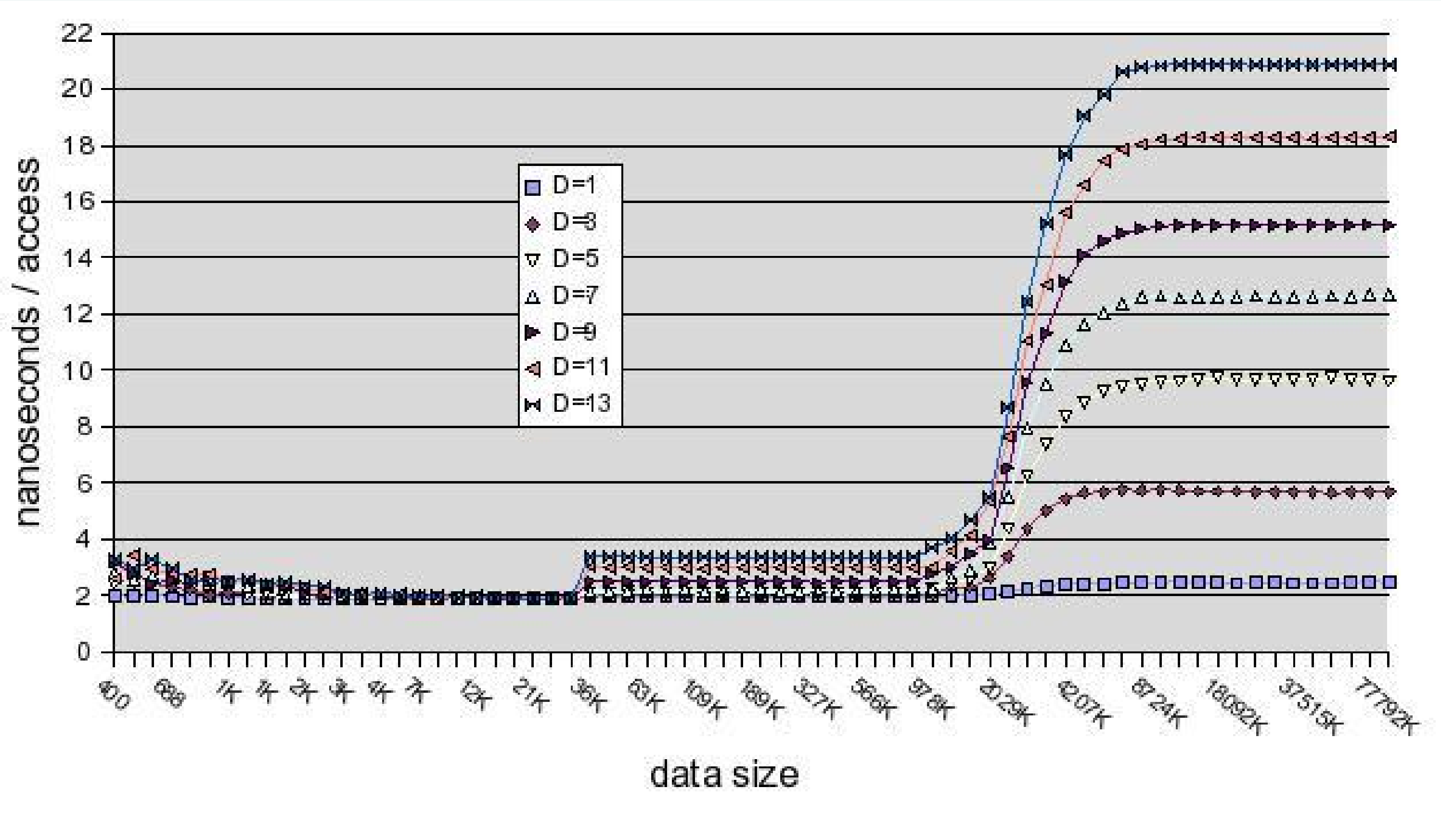
For example, in the IA-32 microprocessors, a 64-bit register stores a counter that is incremented once every CPU clock signal

The `rdtsc` Assembly language instruction reads the value in this register and stores it into the `eax` and `edx` general-purpose registers

Thus, all we need is a bit of *gcc*'s extended inline Assembly magic:

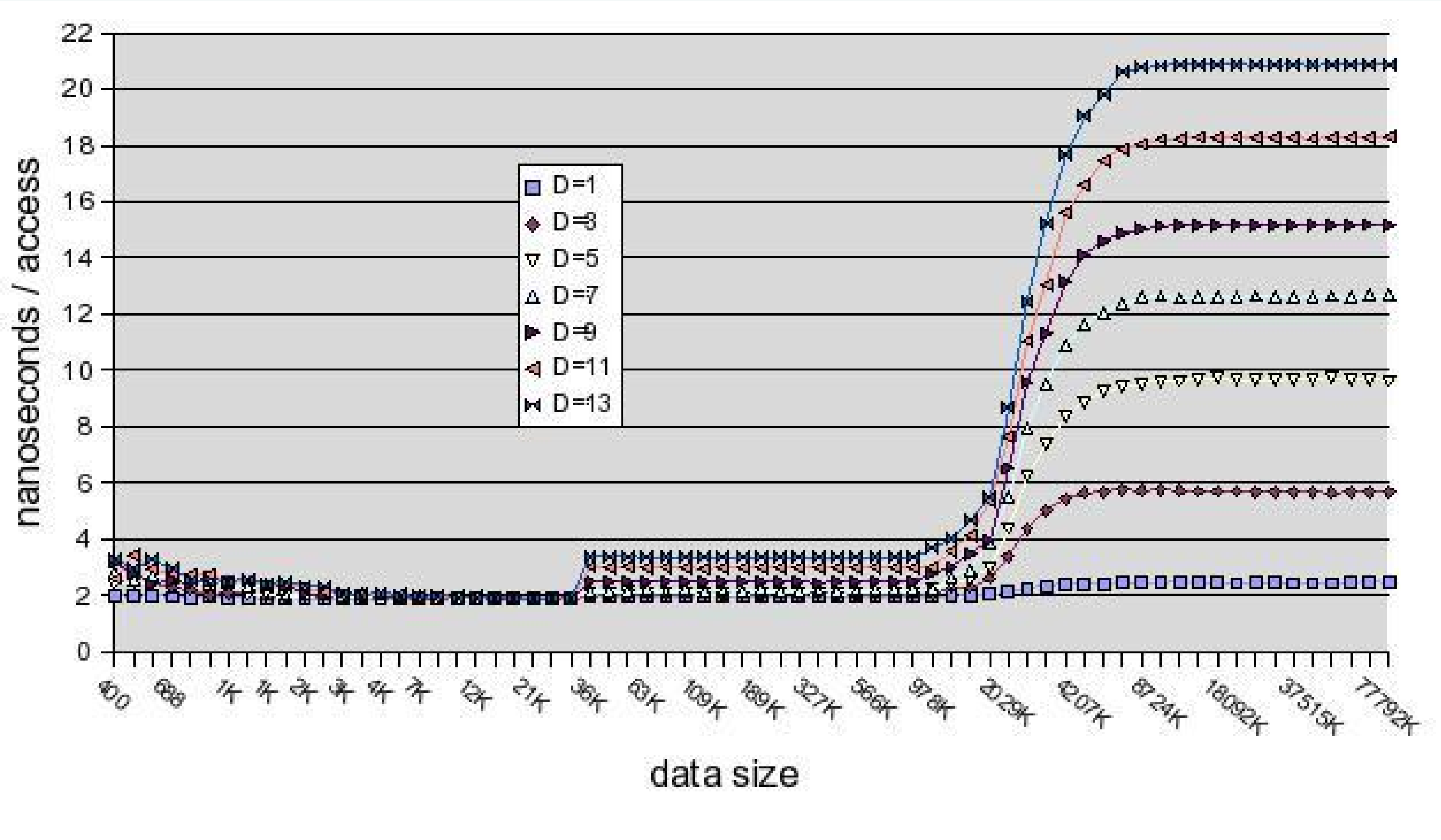
```
#define __rdtscll(val) asm volatile("rdtsc" :  "=A" (val))
```

Making sense out of benchmark values



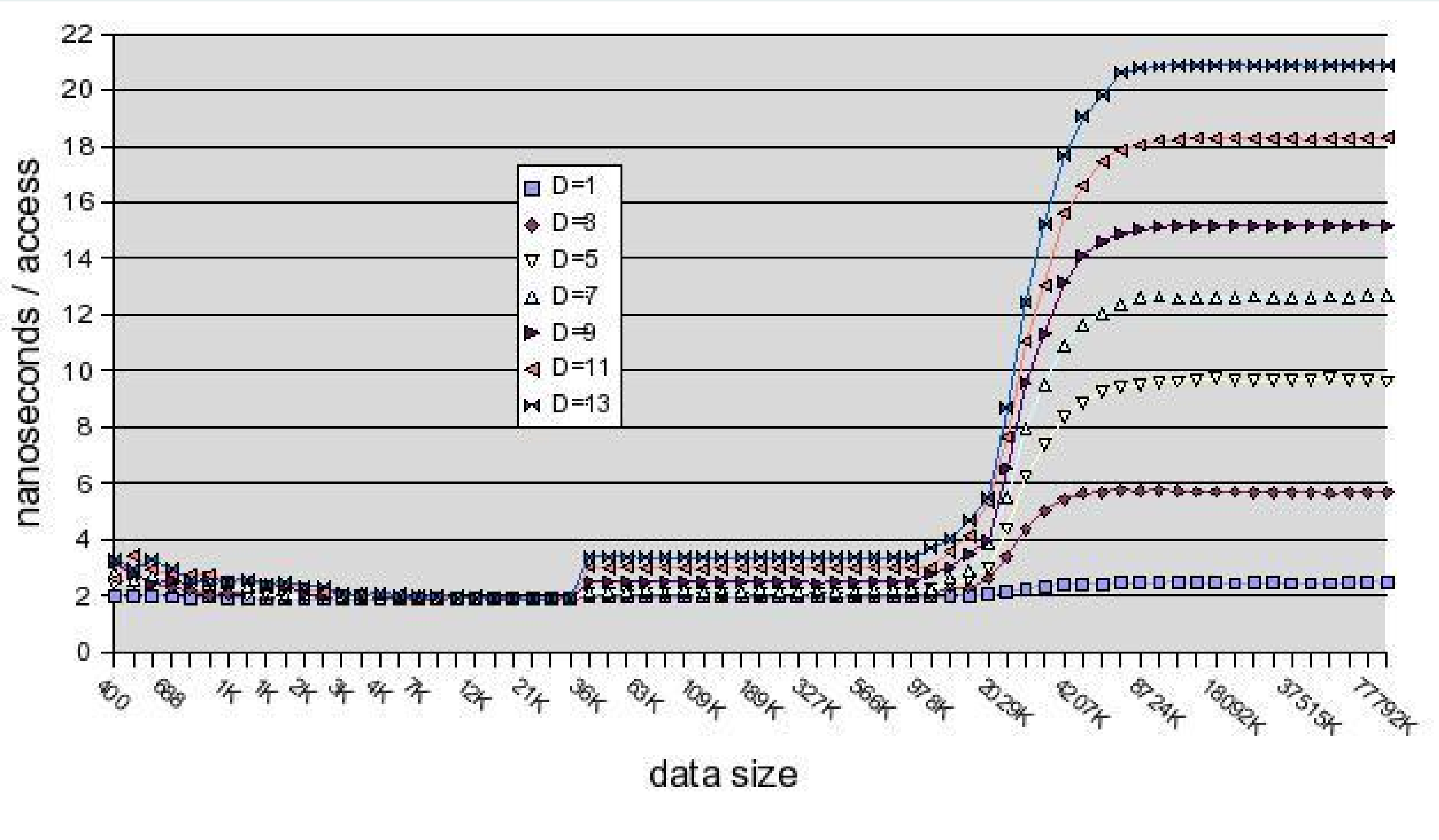
- Each curve is a different delta `d` (cache line=32 byte)

Making sense out of benchmark values



- Each curve is a different delta `d` (cache line=32 byte)
- At smallest data sizes, caches are not fully exploited

Making sense out of benchmark values



- Each curve is a different delta `d` (cache line=32 byte)
- At smallest data sizes, caches are not fully exploited
- “Knees” at cache size boundaries (L1=32 KB, L2=2 MB)

Cache-friendly access to data

Ordering of data accesses does matter. A lot.

Cache-friendly access to data

Ordering of data accesses does matter. A lot.

For example, matrices are usually stored in memory row by row.

Cache-friendly access to data

Ordering of data accesses does matter. A lot.

For example, matrices are usually stored in memory row by row.
We can sequentially scan the elements of a matrix in two ways:

Row by row:

```
for (row=0; row<n; ++row)
    for (col=0; col<n; ++col)
        x[row][col] *= c;
```

Column by column:

```
for (col=0; col<n; ++col)
    for (row=0; row<n; ++row)
        x[row][col] *= c;
```

Cache-friendly access to data

Ordering of data accesses does matter. A lot.

For example, matrices are usually stored in memory row by row.
We can sequentially scan the elements of a matrix in two ways:

Row by row:

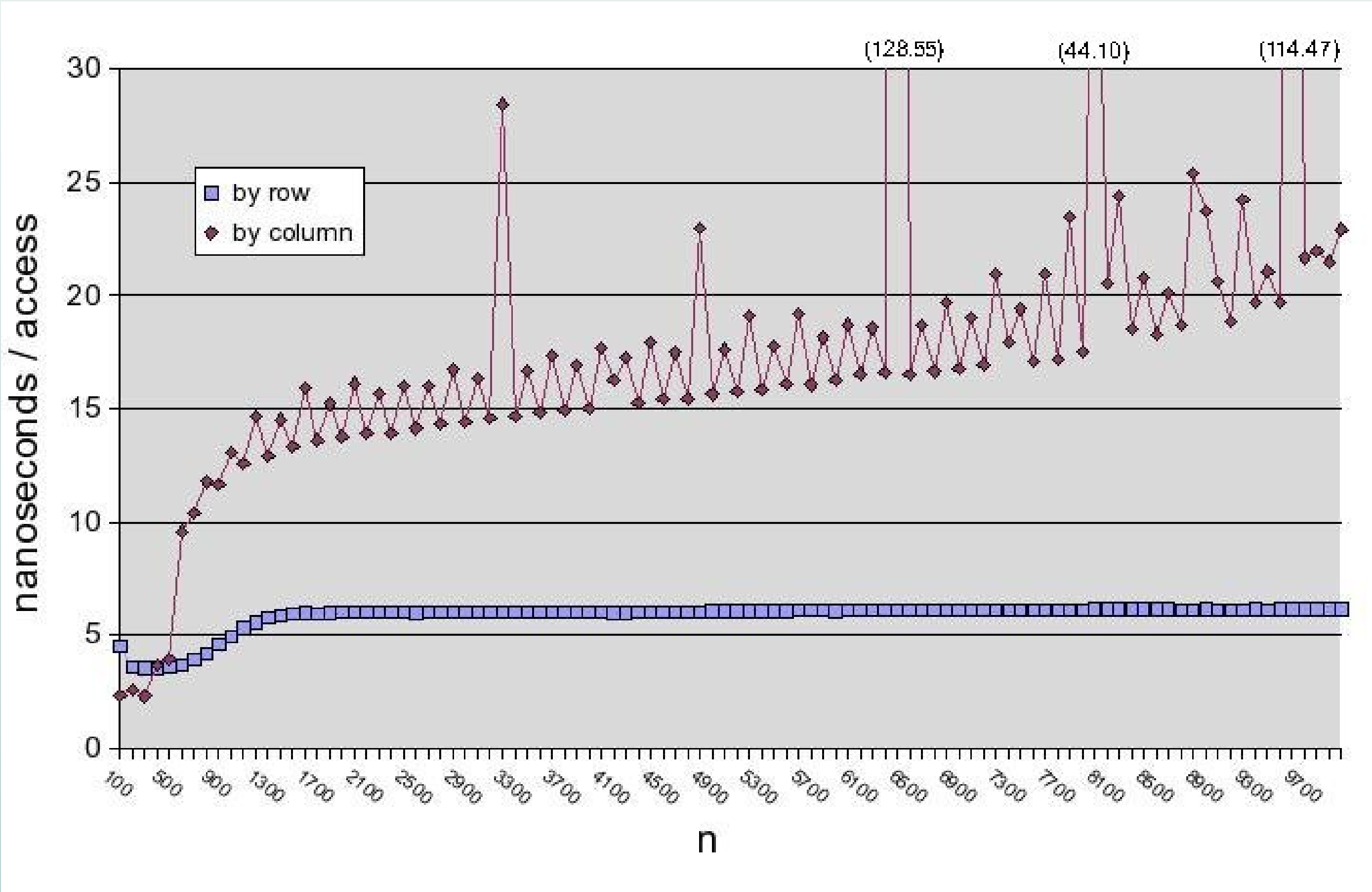
```
for (row=0; row<n; ++row)
    for (col=0; col<n; ++col)
        x[row][col] *= c;
```

Column by column:

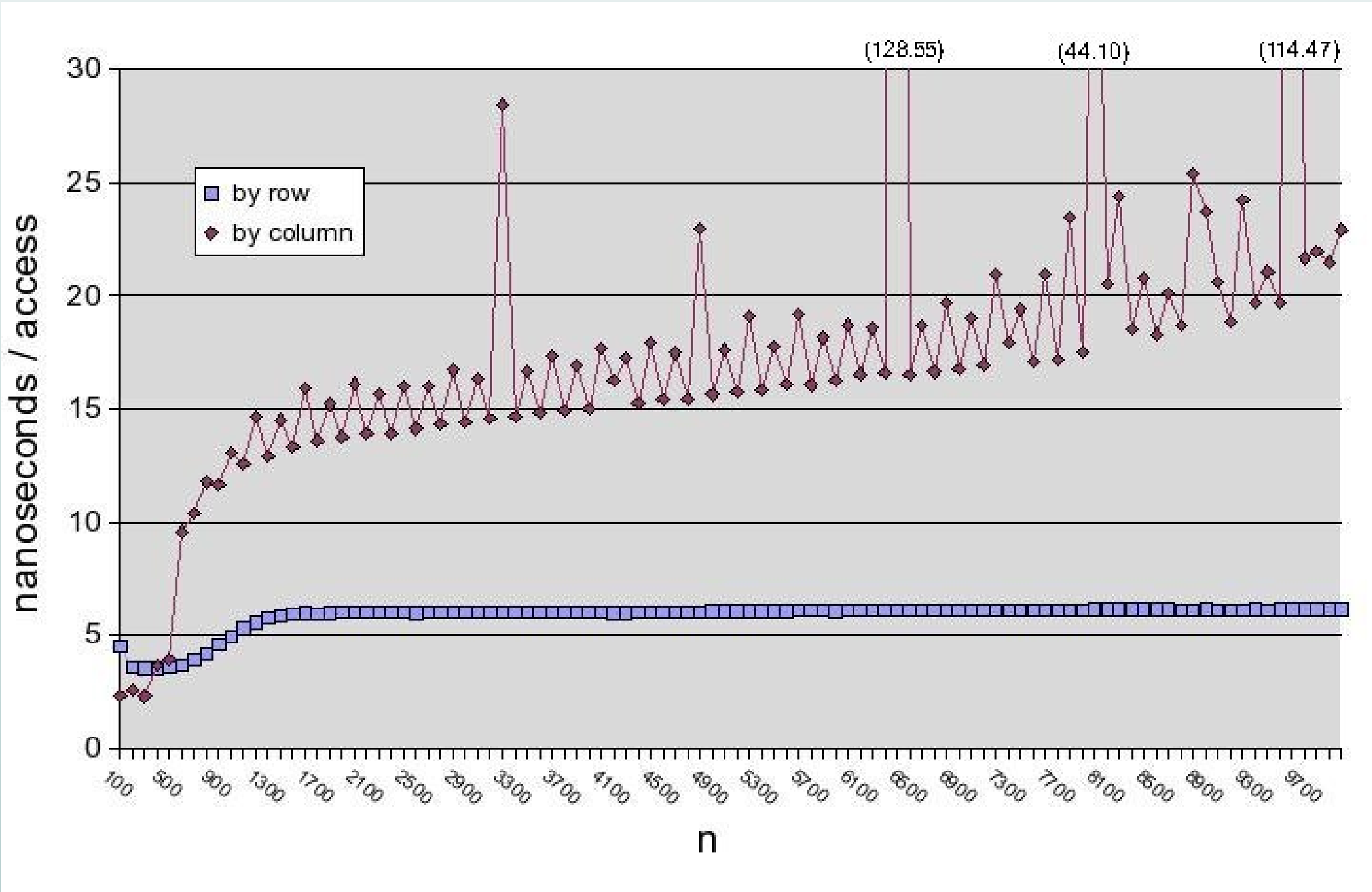
```
for (col=0; col<n; ++col)
    for (row=0; row<n; ++row)
        x[row][col] *= c;
```

Although these code fragments are functionally equivalent, their execution times differ significantly

Execution times of “row by row” and “column by column” scannings

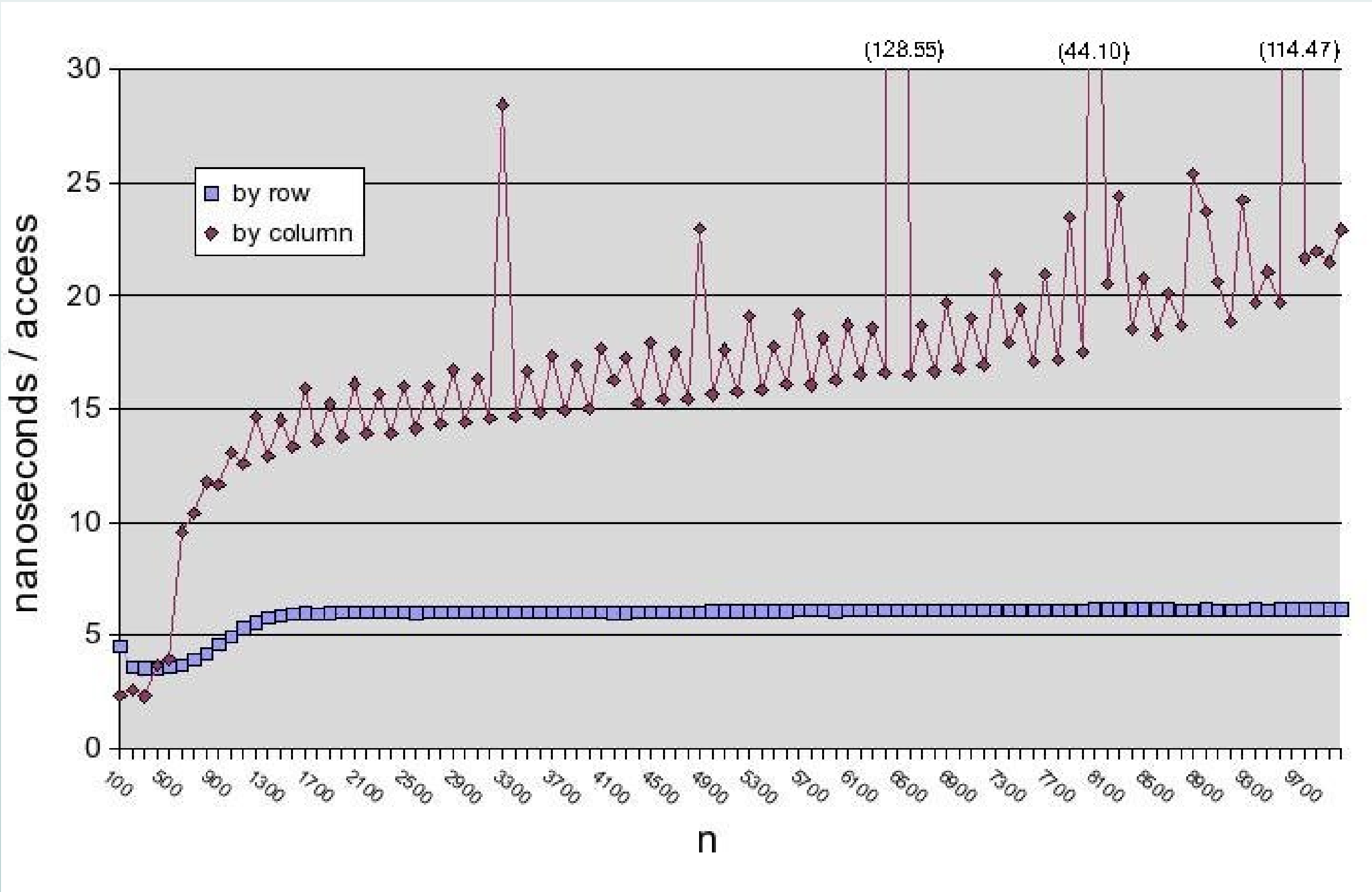


Execution times of “row by row” and “column by column” scannings



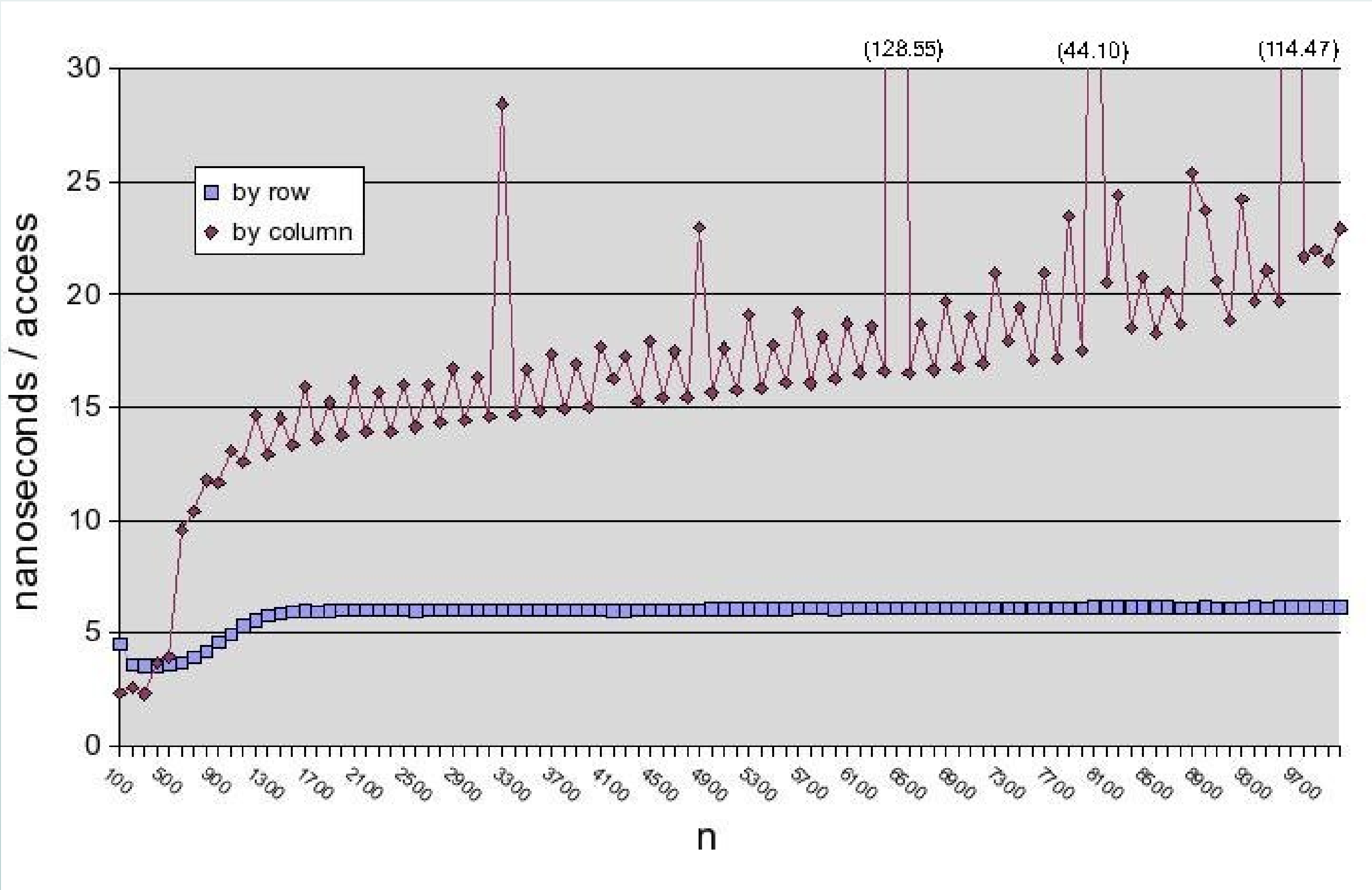
- In the row by row scanning, access time to each element is constant

Execution times of “row by row” and “column by column” scannings



- In the row by row scanning, access time to each element is constant
- In the column by column scanning, access time grows with matrix size

Execution times of "row by row" and "column by column" scannings



- In the row by row scanning, access time to each element is constant
- In the column by column scanning, access time grows with matrix size
- In the column by column scanning there are high peaks when matrix size `n` is a multiple of some cache size parameter (`n` equal to 3200, 4800, 6400, 8000, 9600. . .)

Field reordering in large structures

A well-established optimization technique consists in reordering the fields in large structures according to their access frequencies

Field reordering in large structures

A well-established optimization technique consists in reordering the fields in large structures according to their access frequencies

By putting the most frequently accessed fields near the head of the structure, we raise the likelihood of having the corresponding memory cells mirrored in the hardware caches

Field reordering in large structures

A well-established optimization technique consists in reordering the fields in large structures according to their access frequencies

By putting the most frequently accessed fields near the head of the structure, we raise the likelihood of having the corresponding memory cells mirrored in the hardware caches

It is quite difficult to evaluate the performance gain obtained by this rule

Field reordering in large structures

A well-established optimization technique consists in reordering the fields in large structures according to their access frequencies

By putting the most frequently accessed fields near the head of the structure, we raise the likelihood of having the corresponding memory cells mirrored in the hardware caches

It is quite difficult to evaluate the performance gain obtained by this rule

However, sticking to it should never make any harm

Field reordering in Linux

For example, the first fields of the process descriptor in Linux 2.6.15 are:

```
struct task_struct {
    volatile long state;
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;
    unsigned long ptrace;
    int lock_depth;
    int prio, static_prio;
    struct list_head run_list;
    [...]
}
```

Field reordering in Linux

For example, the first fields of the process descriptor in Linux 2.6.15 are:

```
struct task_struct {
    volatile long state;
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;
    unsigned long ptrace;
    int lock_depth;
    int prio, static_prio;
    struct list_head run_list;
    [...]
}
```

First cache line (32 bytes)

Prefetching

Today's CPUs include instructions for *prefetching*, that is, moving data in a cache before the data itself is effectively required

Prefetching

Today's CPUs include instructions for *prefetching*, that is, moving data in a cache before the data itself is effectively required

For example, IA-32 CPUs include some `prefetch`*X* Assembly instructions that may induce the CPU to speculatively access a memory location

Prefetching

Today's CPUs include instructions for *prefetching*, that is, moving data in a cache before the data itself is effectively required

For example, IA-32 CPUs include some `prefetch`*X* Assembly instructions that may induce the CPU to speculatively access a memory location

A typical scenario consists of scanning the elements of a list. For instance, the Linux macro `list_for_each` yields the following code:

```
for (pos = head->next;
     pos != head; pos = pos->next)
```

Prefetching

Today's CPUs include instructions for *prefetching*, that is, moving data in a cache before the data itself is effectively required

For example, IA-32 CPUs include some `prefetch`*X* Assembly instructions that may induce the CPU to speculatively access a memory location

A typical scenario consists of scanning the elements of a list. For instance, the Linux macro `list_for_each` yields the following code:

```
for (pos = head->next;  prefetch(pos->next),
     pos != head; pos = pos->next)
```

Prefetching

Today's CPUs include instructions for *prefetching*, that is, moving data in a cache before the data itself is effectively required

For example, IA-32 CPUs include some `prefetch`*X* Assembly instructions that may induce the CPU to speculatively access a memory location

A typical scenario consists of scanning the elements of a list. For instance, the Linux macro `list_for_each` yields the following code:

```
for (pos = head->next;  prefetch(pos->next),
     pos != head; pos = pos->next)
```

The `prefetch` macro is a hint for the CPU to read in advance the memory location at `pos->next`; the list element referenced by `pos->next` will be accessed in the next iteration of the loop

Explicit prefetching is dangerous!

You should never add `prefetch` instructions blindly: you must carefully analyze and benchmark your code

Explicit prefetching is dangerous!

You should never add `prefetch` instructions blindly: you must carefully analyze and benchmark your code:

- Today's CPUs perform *speculative accesses* to the DRAM so as to prefetch data in the hardware cache transparently

Explicit prefetching is dangerous!

You should never add `prefetch` instructions blindly: you must carefully analyze and benchmark your code:

- Today's CPUs perform *speculative accesses* to the DRAM so as to prefetch data in the hardware cache transparently
- The `prefetch` instructions have a cost: for instance, ≈ 3.9 nsec on an Intel Pentium 4 at 2.0 GHz, and ≈ 0.6 nsec on an Intel Pentium M at 1.6 GHz

Explicit prefetching is dangerous!

You should never add `prefetch` instructions blindly: you must carefully analyze and benchmark your code:

- Today's CPUs perform *speculative accesses* to the DRAM so as to prefetch data in the hardware cache transparently
- The `prefetch` instructions have a cost: for instance, ≈ 3.9 nsec on an Intel Pentium 4 at 2.0 GHz, and ≈ 0.6 nsec on an Intel Pentium M at 1.6 GHz
- The `prefetch` instructions steal space in the instruction cache; this is particularly bad if the instructions are inside a small loop

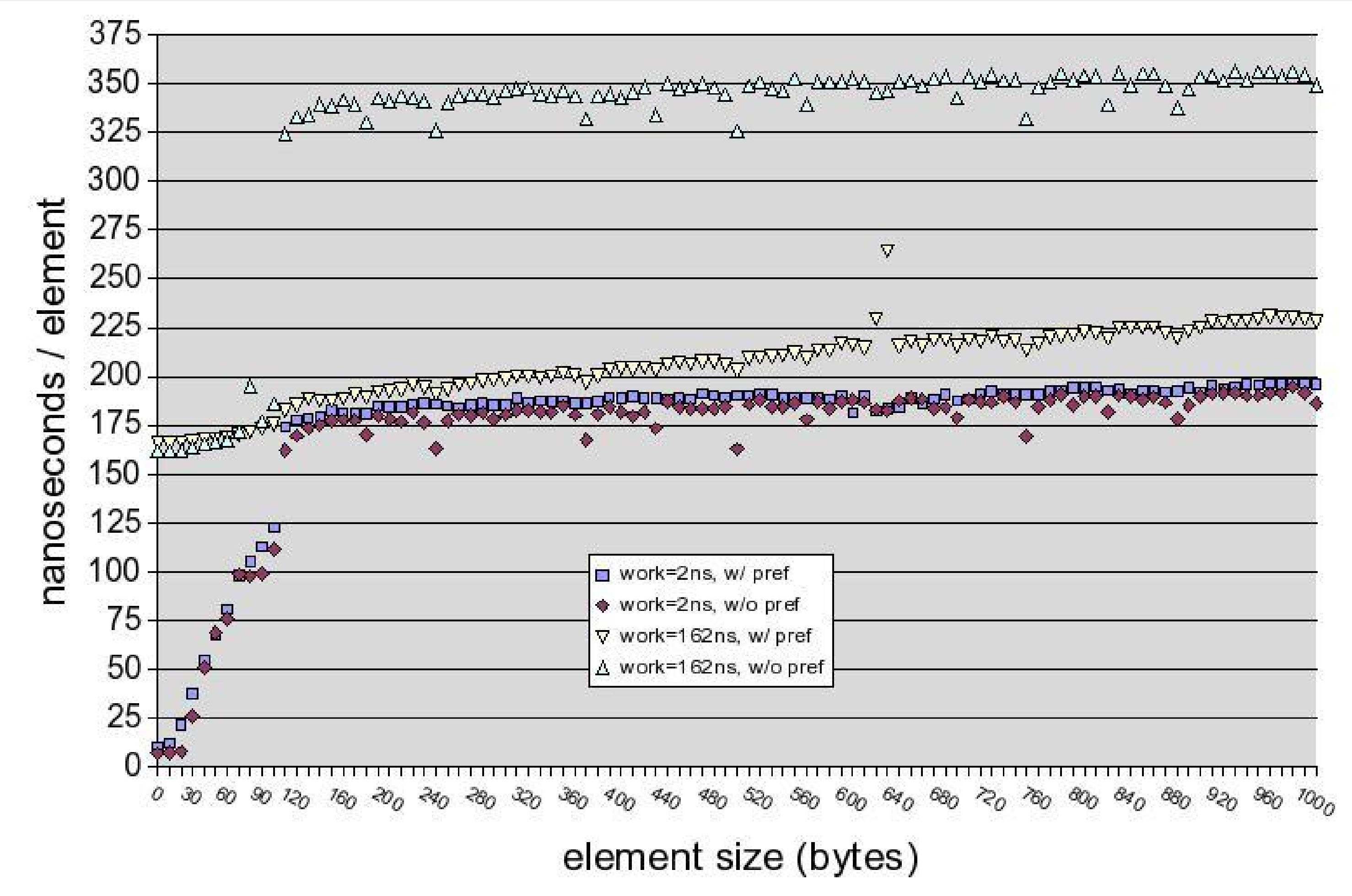
Explicit prefetching is dangerous!

You should never add `prefetch` instructions blindly: you must carefully analyze and benchmark your code:

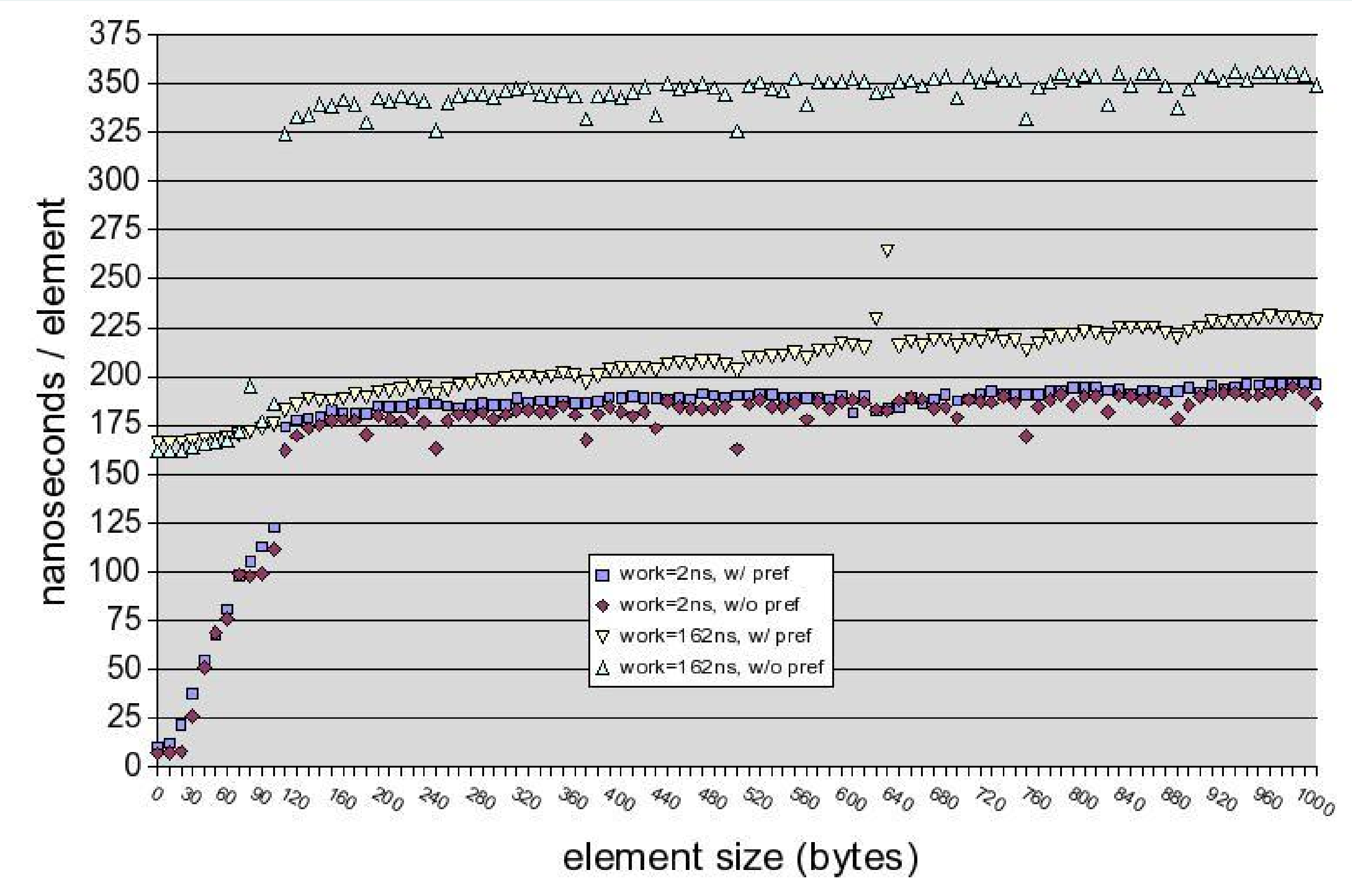
- Today's CPUs perform *speculative accesses* to the DRAM so as to prefetch data in the hardware cache transparently
- The `prefetch` instructions have a cost: for instance, ≈ 3.9 nsec on an Intel Pentium 4 at 2.0 GHz, and ≈ 0.6 nsec on an Intel Pentium M at 1.6 GHz
- The `prefetch` instructions steal space in the instruction cache; this is particularly bad if the instructions are inside a small loop

Thus, by using explicit prefetching you might actually impair your code!

Prefetching on list scanning

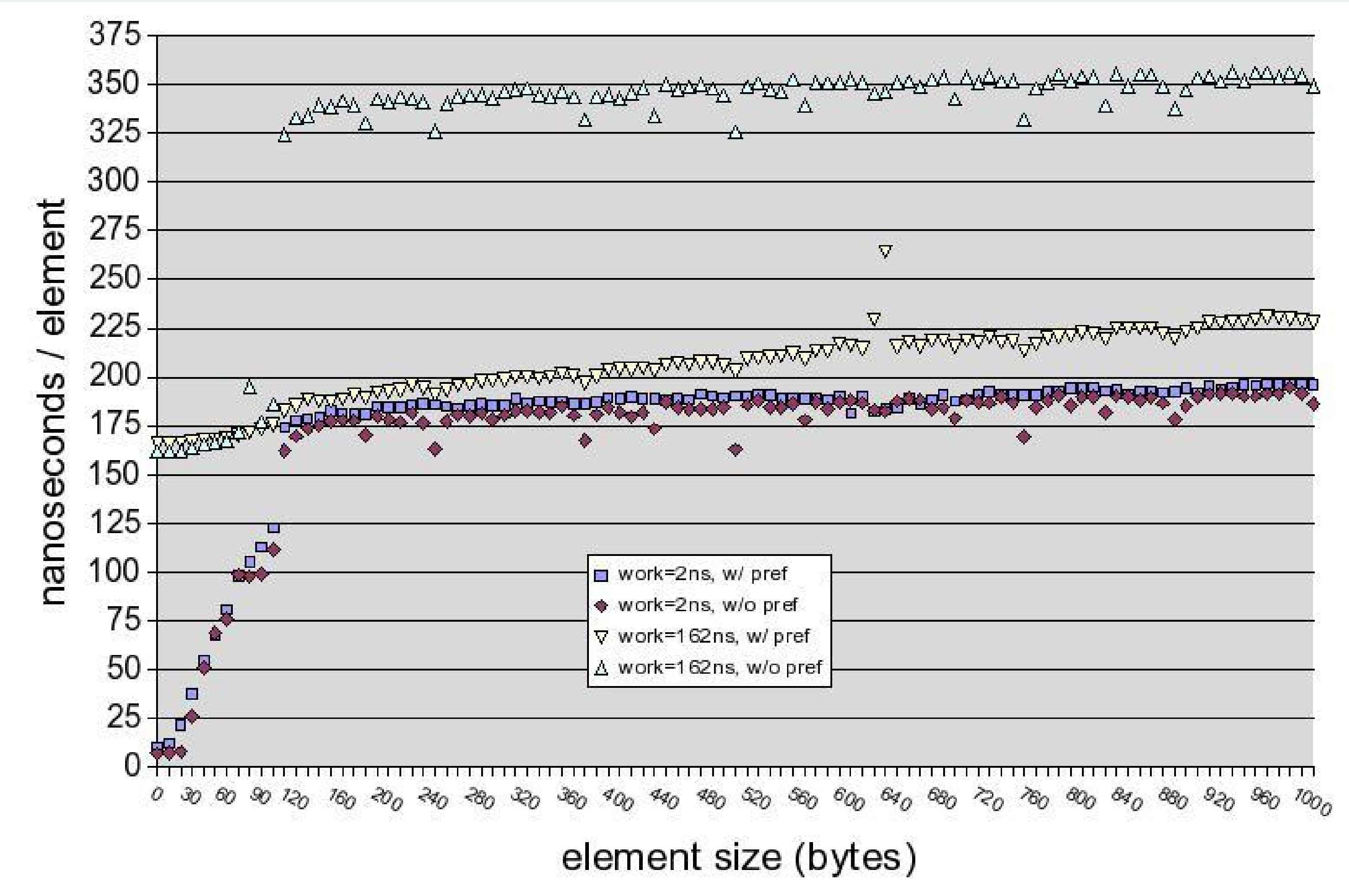


Prefetching on list scanning



- If the time spent in processing each list element is small (2 nsec), scanning without prefetching is faster; time difference is roughly the execution time of the `prefetch` instruction

Prefetching on list scanning



- If the time spent in processing each list element is small (2 nsec), scanning without prefetching is faster; time difference is roughly the execution time of the `prefetch` instruction
- If the time spent in processing each list element is comparable to the DRAM access time, scanning with prefetching is much faster

Branch prediction

In today's CPUs, machine instructions are executed in parallel whenever possible: the pipeline includes many instructions in different stages of execution

Branch prediction

In today's CPUs, machine instructions are executed in parallel whenever possible: the pipeline includes many instructions in different stages of execution

Branch prediction can be viewed as another caching technique:

- It consists of guessing the path that will be taken by a conditional jump well before the outcome of the jump condition has been determined

Branch prediction

In today's CPUs, machine instructions are executed in parallel whenever possible: the pipeline includes many instructions in different stages of execution

Branch prediction can be viewed as another caching technique:

- It consists of guessing the path that will be taken by a conditional jump well before the outcome of the jump condition has been determined

- It is usually either *dynamic* (based on the past branches taken by that particular jump) or *static* (if no branch history is available)

Branch prediction

In today's CPUs, machine instructions are executed in parallel whenever possible: the pipeline includes many instructions in different stages of execution

Branch prediction can be viewed as another caching technique:

- It consists of guessing the path that will be taken by a conditional jump well before the outcome of the jump condition has been determined
- It is usually either *dynamic* (based on the past branches taken by that particular jump) or *static* (if no branch history is available)

A well-predicted jump is very cheap (typically, zero or one CPU cycles), while a mispredicted jump is very costly, because the instructions of the path not taken must be removed from the pipeline

Static branch prediction

The rule for *static branch prediction* in IA-32 CPUs is simple

Static branch prediction

The rule for *static branch prediction* in IA-32 CPUs is simple:

- Any forward jump (positive offset) is predicted not to be taken

Forward jump:

```
    jxx l0
    (likely branch)
l0:
    (unlikely branch)
```

Static branch prediction

The rule for *static branch prediction* in IA-32 CPUs is simple:

- Any forward jump (positive offset) is predicted not to be taken
- Any backward jump (negative offset) is predicted to be taken

Forward jump:

```
    jxx l0
    (likely branch)
l0:
    (unlikely branch)
```

Backward jump:

```
l1:
    (likely branch)
    jxx l1
    (unlikely branch)
```

Exploiting the static branch prediction

In order to generate code that is aware of the static branch prediction rule, Linux defines the following macros:

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

Exploiting the static branch prediction

In order to generate code that is aware of the static branch prediction rule, Linux defines the following macros:

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

They rely on the `__builtin_expect` *gcc*'s extension, which forces the compiler to generate code optimized for the case in which `x` is, respectively, expected to be 0 or 1

Exploiting the static branch prediction

In order to generate code that is aware of the static branch prediction rule, Linux defines the following macros:

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

They rely on the `__builtin_expect` *gcc*'s extension, which forces the compiler to generate code optimized for the case in which `x` is, respectively, expected to be 0 or 1

On IA-32, "`if (unlikely(current->state == TASK_STOPPED))`" generates either a branch-if-true forward jump or a branch-if-false backward jump

Static branch prediction with *gcc* 4.1

The *gcc* 4.1 compiler allows you to tag the branches in your code without "wild guesses"

Static branch prediction with *gcc* 4.1

The *gcc* 4.1 compiler allows you to tag the branches in your code without “wild guesses”:

1. Compile your program by enabling code optimization (for instance, `-O2`) and by using the `-fprofile-arcs` option to *gcc*

Static branch prediction with *gcc* 4.1

The *gcc* 4.1 compiler allows you to tag the branches in your code without “wild guesses”:

1. Compile your program by enabling code optimization (for instance, `-O2`) and by using the `-fprofile-arcs` option to *gcc*

2. Run your program in a typical scenario

Static branch prediction with *gcc* 4.1

The *gcc* 4.1 compiler allows you to tag the branches in your code without “wild guesses”:

1. Compile your program by enabling code optimization (for instance, `-O2`) and by using the `-fprofile-arcs` option to *gcc*

2. Run your program in a typical scenario

3. Compile again your program by using the same optimization flags as in step 1 (`-O2`) and by using the `-fbranch-probabilities` option to *gcc*

Static branch prediction with *gcc* 4.1

The *gcc* 4.1 compiler allows you to tag the branches in your code without “wild guesses”:

1. Compile your program by enabling code optimization (for instance, `-O2`) and by using the `-fprofile-arcs` option to *gcc*

2. Run your program in a typical scenario

3. Compile again your program by using the same optimization flags as in step 1 (`-O2`) and by using the `-fbranch-probabilities` option to *gcc*

All done! The compiler will generate code optimized for the branches effectively taken in the run at step 2

Out-of-section branching

Linux stretches the concept of optimizing branch prediction a bit further

Out-of-section branching

Linux stretches the concept of optimizing branch prediction a bit further

Branches corresponding to exceptional or anomalous conditions are not stored in the section of normal code; instead, they are put in a different section

Out-of-section branching

Linux stretches the concept of optimizing branch prediction a bit further

Branches corresponding to exceptional or anomalous conditions are not stored in the section of normal code; instead, they are put in a different section:

```
        jxx oops
        (normal, likely branch)
        .section .fixup
oops:
        (exceptional, unlikely branch)
        .previous
```

Out-of-section branching in C

A sort of out-of-section branching can be implemented directly in C by means of a *gcc*'s extension:

```
void oops_handler(void)
        __attribute__((section(".exceptions")));
```

[. . .]

```
if (unlikely(oops_condition))
    oops_handler();
```

Out-of-section branching in C

A sort of out-of-section branching can be implemented directly in C by means of a *gcc*'s extension:

```
void oops_handler(void)
        __attribute__((section(".exceptions")));
```

[. . .]

```
if (unlikely(oops_condition))
    oops_handler();
```

To specify the address of the new section of code, pass an option to the linker:

in *ld*: `ld --section-start .exceptions=0x08049000 prog.o`

Out-of-section branching in C

A sort of out-of-section branching can be implemented directly in C by means of a *gcc*'s extension:

```
void oops_handler(void)
        __attribute__((section(".exceptions")));
```

[...]

```
if (unlikely(oops_condition))
    oops_handler();
```

To specify the address of the new section of code, pass an option to the linker:

in *ld*: `ld --section-start .exceptions=0x08049000 prog.o`
in *gcc*: `gcc -O2 -Wl,--section-start,.exceptions=0x08049000 prog.c`

Function reordering in gcc 4.1

Moving functions across different sections might be beneficial for the program's performances

Function reordering in gcc 4.1

Moving functions across different sections might be beneficial for the program's performances

If all the most frequently accessed functions are placed in some section of code (let's say, `text.hot`) while all the less frequently accessed functions are placed in another section (`text.unlikely`), there would be a higher number of hits versus misses in the hardware caches

Function reordering in gcc 4.1

Moving functions across different sections might be beneficial for the program's performances

If all the most frequently accessed functions are placed in some section of code (let's say, `text.hot`) while all the less frequently accessed functions are placed in another section (`text.unlikely`), there would be a higher number of hits versus misses in the hardware caches

This can be done automatically by the *gcc* 4.1 compiler! You must enable optimization, use the `-fprofile-arcs` option, and compile twice, as described earlier

Function reordering in the kernel

Performances of User Mode programs improve with function reordering: shouldn’t this hold also for the kernel?

Function reordering in the kernel

Performances of User Mode programs improve with function reordering: shouldn't this hold also for the kernel?

Arjan van de Ven (a well-known kernel hacker working at Intel) is currently developing a Linux patch for the x86_64 architecture precisely with this goal

Function reordering in the kernel

Performances of User Mode programs improve with function reordering: shouldn't this hold also for the kernel?

Arjan van de Ven (a well-known kernel hacker working at Intel) is currently developing a Linux patch for the x86_64 architecture precisely with this goal

In particular, the patch makes use some features of the *gcc* 4.1 compiler to:

- Move the kernel image so that it starts at physical address 2 MB (that is, at the beginning of a physical memory area mapped by a large TLB entry)

Function reordering in the kernel

Performances of User Mode programs improve with function reordering: shouldn't this hold also for the kernel?

Arjan van de Ven (a well-known kernel hacker working at Intel) is currently developing a Linux patch for the x86_64 architecture precisely with this goal

In particular, the patch makes use some features of the *gcc* 4.1 compiler to:

- Move the kernel image so that it starts at physical address 2 MB (that is, at the beginning of a physical memory area mapped by a large TLB entry)
- Move code tagged as `unlikely()` in a separate section

Function reordering in the kernel

Performances of User Mode programs improve with function reordering: shouldn't this hold also for the kernel?

Arjan van de Ven (a well-known kernel hacker working at Intel) is currently developing a Linux patch for the x86_64 architecture precisely with this goal

In particular, the patch makes use some features of the *gcc* 4.1 compiler to:

- Move the kernel image so that it starts at physical address 2 MB (that is, at the beginning of a physical memory area mapped by a large TLB entry)
- Move code tagged as `unlikely()` in a separate section
- Put the most frequently accessed functions starting at the 2 MB boundary

Function reordering in the kernel

Performances of User Mode programs improve with function reordering: shouldn't this hold also for the kernel?

Arjan van de Ven (a well-known kernel hacker working at Intel) is currently developing a Linux patch for the x86_64 architecture precisely with this goal

In particular, the patch makes use some features of the *gcc* 4.1 compiler to:

- Move the kernel image so that it starts at physical address 2 MB (that is, at the beginning of a physical memory area mapped by a large TLB entry)
- Move code tagged as `unlikely()` in a separate section
- Put the most frequently accessed functions starting at the 2 MB boundary

In some *lmbench* benchmarks, Arjan claims a 10% gain in performances!!

Conclusions

- Someone is still arguing that Unix's heritage makes Linux obsolete, but let's face it: Linux is on the bleeding edge of software development

Conclusions

- Someone is still arguing that Unix's heritage makes Linux obsolete, but let's face it: Linux is on the bleeding edge of software development

- Reading Linux source code may expose a programmer to new, risky, and amazing ideas

Conclusions

- Someone is still arguing that Unix's heritage makes Linux obsolete, but let's face it: Linux is on the bleeding edge of software development
- Reading Linux source code may expose a programmer to new, risky, and amazing ideas
- Finally, Linux might help you in many ways. . . even if you don't run it!