No more free lunch, maybe a pie for free?

How to survive computing paradigm shift.

Adam Strzelecki

2008-06-16

We cannot count on free "performance lunch" anymore[1], but how about at least a pie for free? Do we need to throw all our old source-code into the trash bin and start over again?

Certainly, not. We may think of our old software as a zombie of the new multi-core era. Still there is a way to make the zombie walk, even walk faster. Of course we will need to rewrite our code sometime, but we may postpone this nasty need for a while.

The computing paradigm shift is now a fact we need to learn how to live with. It is unquestionable that processor manufacturers hit the barrier of 3 GHz. There pretty many news about spinning the CPU up to 5 GHz or so, but do not try to do so at home unless you got liquid nitrogen cylinder around. Over 3 GHz heat emission grows unreasonably making the CPU economically worthwhile.

So the only sensible way is now horizontal performance improvement, doubling the number of processing units. This means we can expect soon 128 core CPUs. Oh, wait they are here already; nVidia GeForce 9 series are perfect example of 128 core streaming processor.

Now how to keep up with things that happen so fast. First we need focus more on performance of our code. Something that was not important before, now cannot be ignored.

[1] Sutter, H. 2005. "The free lunch is over: A fundamental turn toward concurrency in software," Dr. Dobb's Journal, 30(3), http://www.gotw.ca/publications/concurrency-ddj.htm

1. Is your code optimized enough for sequential computing?

We once got a comfort to write lazy programs that were revived by brand new processors after one or two years. This made us believe that optimization is just a waste of time and we may write our algorithms, so the do what they are supposed to do without a care about their robustness, just to be out of the work earlier and grab cold beer we were waiting for all day.

Now there is a time to dust off old procedures, profile the software and catch the performance suckers. I am sure we already asked ourselves whether we have used the lowest computing complexity algorithm. But there are some other things we may do about weak performance of our old sequential code.

1.1. Is your code/compiler using latest instructions sets?

About 10 years ago Intel processor instructions set has grown with fancy new thing called MMX, this was somehow a consequence of pipelining introduced with Pentium processors. MMX instructions were SIMD (single instruction, multiple data) executing several arithmetical operations within single instruction. Switching to MMX gave applications visible performance gain. However MMX had one significant weakness, it was only dealing with integer registers, soon afterwards AMD released its own 3DNow!, then Intel again addressed MMX shortcomings with Pentium III SSE dealing with full set of floating point registers.

This is pretty old story that happens to be not really familiar to the programmers. Of course most of them have heard about MMX, SSE, SSE2 & 3, but "hearing" was the closest encounter ever.

Maybe it is time to get another encounter with those instructions. Look into your compiler's manual for SSE, or maybe there is a new release of your favourite compiler that generates SSE. If that is not enough you may try some aggressive optimizing compilers such as LLVM-GCC or Intel Compiler.

Try to play with inline assembler too. If you do a lot of compression, post-processing, real-time effects, knowing assembler may be a virtue.

Being Java or Ruby programmer it may be a time for you to upgrade your VM or interpreter distribution. Java 1.6 scores 30% boost over 1.5 in server benchmarks, while upcoming Ruby 1.9 with YARV VM smokes old MRV 1.8 away with 2× - 10× boost. There are other notable performance boosters, such as LLVM-GCC for C/C++/Obj-C code and Squirellfish for JavaScript.

As you see there is more buzz now about performance improved releases of old software, and everything just because buying new CPU is not enough anymore.

1.2. Others can do it better

Once you got new outstanding compiler you may ask yourself what can I do more do get more juice? Most of the computer software share the same ideas, same algorithms under the hood. Why not to take something that was done by somebody else, and serves well for the others, a library.

Chip engineers work closely together with skilled programmers that prepare highly optimized and well designed libraries containing commonly used functions, such as math trigonometry, matrix operations, even discreet transformations coded using latest CPU instruction sets.

Most notable pair is Intel® Integrated Performance Primitives 5.3[2] & AMD Performance Library (APL)[3]. You may try one of those if your application is suppose to use lot of JPEG processing, matrix operations, and so.

Also it is quite common that such libraries are making already good use of multiple core CPUs, so if your code spends most of its time in functions available in those libraries it is certainly worth to try those libraries out.

I shall mention that it is tempting to rewrite all over again some common runtime functions, just because we could not find `strcasecmp` while it was called `stricmp` on our platform. This is really nasty way. Let me give you an example. Imagine we miss `strlen` somehow, just because we were lazy enough not to check how to include `string.h`. So let write our `strlen`:

```
size_t strlen (const char *str) {
  const char *cstr = str;
  while (*cstr) cstr++;
  return cstr - str;
}
```

That was simple and works as desired! But have a look at the function below:

```
size_t strlen (const char *str) {
  const unsigned long int *longword_ptr = (unsigned long int *) str;
  unsigned long int longword,
    magic_bits = 0x7efefeffL,
    himagic = 0x80808080L,
    lomagic = 0x01010101L;
  for (;;) {
    longword = *longword_ptr++;
    if (((longword - lomagic) & himagic) != 0) {
      const char *cp = (const char *) (longword_ptr - 1);
      if (cp[0] == 0) return cp - str;
      if (cp[1] == 0) return cp - str + 1;
      if (cp[2] == 0) return cp - str + 2;
      if (cp[3] == 0) return cp - str + 3;
    }
  }
}
```

Would you tell that it can run 4×times faster that our simple function[4]? Well it does, taken from GLIBC[5] it does fetch & check 4 bytes at once, while our simple `strlen` does only check 1 byte in the same moment.

As you see runtime developers use pretty many tricks to speed up their code, knowing the platform internals. Remember then, do NOT try to rewrite the code that was done and polished already by "smarter" developers.

[2]Intel® Integrated Performance Primitives (Intel® IPP) is an extensive library of multi-core-ready, highly optimized software functions for multimedia data processing, and communications applications by Intel `http://www.intel.com/cd/software/products/asmo-na/eng/302910.htm`

[3]AMD Performance Library (APL) is now open-source framework by AMD `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~123872,00.html`

[4]Assuming that we are dealing with multi-byte null-terminated ASCII strings.

[5]The GNU C library is used as the C library in the GNU system and most systems with the Linux kernel. `strlen` function taken from GLIBC `string/strlen.c`.

1.3. Almighty compiler

Well the compiler is unfortunately not almighty at all, so do not believe those who say it is. Sometimes putting too much trust into compiler is a sinful way. I know some stubborn developers saying they priers ["Compiler will anyway optimize everything"], wondering why their software works so slow afterwards.

Compiler does not know your intentions, moreover it assumes your intentions are right and tries to do exactly what you state in the code. So if you really write something unreasonably stupid it will not save you from the eternal flame.

There is one important rule when programming; code that is not readable for human being will not be fully understood by the optimizer, even it is syntactically valid.

Using `*(table + i)` instead of `table[i]` may mean the same for you, however it will not be the same for the optimizer, that will not try to touch `*(table + i)` assuming this is awkwardness and it has to be run exactly as it is stated. Using normal (index) notation we allow compiler to use SIMD instructions where possible or even theoretically parallelize some of the loops.

1.4. Don't forget about memory pill

RAM and L1 & L2 cache memory is important factor in overall system performance. While accessing cache memory may take few CPU cycles, access to RAM may take tens or hundreds of CPU cycles.

Most of the compilers try to minimize access to the memory keeping as much as it is possible in the registers, then placing the variables in the memory coupled together lowering the cache miss ratio.

Compilers try also to align data to the CPU (machine) word boundary. 32-bit processor has 32-bit words, so regardless how small the data is, it will always read 32 bits (4 bytes) withing 32-bit alignment in one shot. As a consequence reading 1 byte, 2 bytes or 4 bytes takes exactly the same amount of cycles for 32-bit CPU.

Modern processors access RAM trough L1 & L2 caches. While L1 is usually smaller than 128KB, L2 may be even 2MB. L1 is fastest cache memory, L2 is 2-3 times slower, while still around 10 times faster than RAM.

Cache is divided into cache lines, usually 64 or 32 bytes. Each line is a chunk of cached RAM memory with address starting with multiple of cache size.

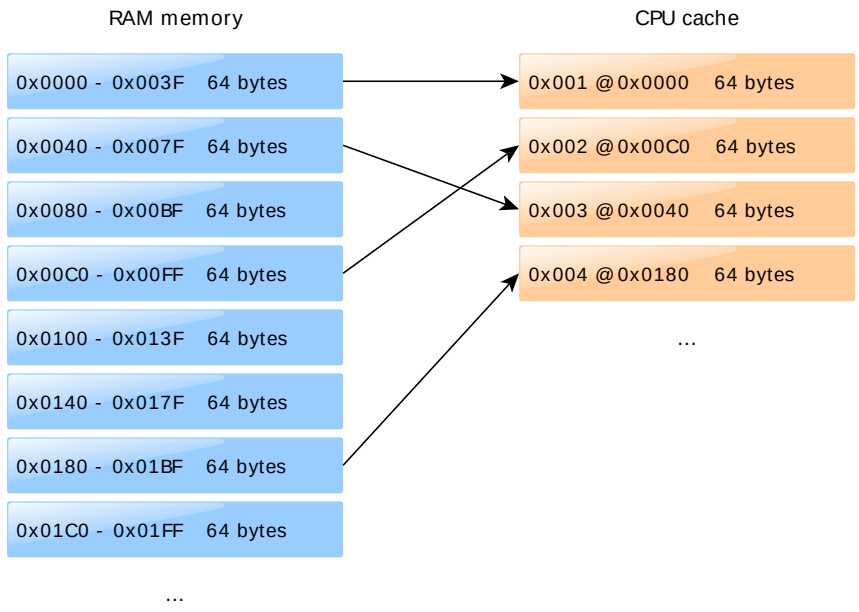

Figure 1. CPU cache

Minimizing cache miss ratio means then coupling the single function variables all together so they use minimal number of cache lines so cache line swap and RAM access occurs seldom.

While new processors do not bring higher clock-rates, they bring bigger on-die cache memories. This means that we can get more performance from our old sequential code, just if we do not spoil memory and cache access.

1.4.1. Array indexing order

Compilers such as C/C++/Fortran always arrange data structures such as arrays specific way. It is important to know this arrangement, whether for two dimensional table rows or columns goes together first.

Choosing wrong indexing order those two

```
void myfun(int mul) {
  int i, j;
  int values[1024][1024];
  /* nasty order */
  for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
      values[i][j] = i * j * mul;
  (...)
}
```

```
void myfun(int mul) {
  int i, j;
  int values[1024][1024];
  /* well done! */
  for (i = 0; i < 1024; i++)
    for(j = 0; j < 1024; j++)
      values[i][j] = i * j * mul;
  (...)
}
```

... may result with lousy performance by misuse of the cache.

1.4.2. Local variable declaration order

Small variables used together that are declared together will use less cache lines than when their declarations are spread.

We often add variable declarations at the end of declaration block as the code grows. As compiler does exactly what we write (unless we turn on some "aggressive" optimization like GCC's O3) it will place the variables on the stack (in the memory) in the order they were declared.

So if we declare `i`, `j`, `k` variables as below.

```
static int i;
static char body[1024];
static int j;
static double x, y, z, r, w = 0;
static int k;
for (i = 0; i < x + y; i++)
  for (j = 0; j < z; j++)
   for (k = 0; k < j; k++)
     (...)
```

We can be sure that `i`, `j`, `k` will be placed finally in different cache lines. Since this efficient cache memory is very limited, especially in multi-process environment, it is much better to:

1. Declare together variables that are used together

2. Declare variables from the smallest to the biggest (in terms of storage type size)

```
static int i, j, k;
static double x, y, z, r, w = 0; /* sizeof(double) >= sizeof(int) */
static char body[1024];          /* I am the biggest */
for (i = 0; i < x + y; i++)
  for (j = 0; j < z; j++)
    for (k = 0; k < j; k++)
      (...)
```

1.4.3. Global, dynamic and stack frame variables

It is worth to use local stack frame variables (local function variables) where possible. First of all allocation (and deallocation) of them takes no time and you do not need to worry to free them. Second important reason is that often compiler can turn such variable into CPU register with instant access time, while global and dynamic variables cannot be stored in the registers.

If you need to use dynamic variables (like because of the one obvious reason; ANSI C requires that local variables have fixed size) is it worth to use `alloca` function instead of regular `malloc`, when you anyway plan to release allocated memory before your function finishes. `alloca` uses stack frame space, and does not require calling `free`, but allocated memory is valid only during the execution of the function where the variable was allocated at.

Heap management functions such as `malloc`, `calloc` or `free` have one important drawback, they consume time. So it is important to limit their usage if possible.

1.4.4. Constant variables

Marking variables to be constant (for example with const C/C++ keyword) is not just a hint for mate programmers to not make any change to this variable's value, but an important hint for the compiler's optimizer.

Normally we often omit const declaration even if we are not planning to change the variable value, so it can be considered constant.

```
void myfunction() {
  char format[] = "My string = %s";
  (...)
}
```

However then we can expect that compiler will generate CPU instructions that will copy `"My string = %s"` into the stack space pointed by `format` on every function call. This is lousy behavior for variable that will not be anyway modified, loosing CPU cycles doing memory copy all over again.

But what will happen if we add const ?

```
void myfunction() {
  const char format[] = "My string = %s";
  (...)
}
```

Compiler will just point `format` (push an address) to the constant read-only memory region (often a `BSS` block) containing `"My string = %s"` loaded once at program startup, so instead expensive memory copy, we just have cheap pointer assignment at function entry.

2. Boosting old sequential code with multi-core

Once our sequential algorithms shine, we may wonder if there is a chance to make use of 2 or 4 cores within our old program.

Fortunately, yes! In most of the cases at least, if you are lucky enough. The obvious way to have good use of multi-core is to run several processes at once. So if we got CoreDuo and we want to do some MP3 compression, we can compress 2 files at once as we were running on 2 single core computers.

Other good example is compilation. We deal with many relatively small files. Build process lunches compiler per each file. Compiling one file is usually not dependent on the other, so we can speed compilation up calling:

```
make -j2
```

Where `-j2` tells `make` to run two commands (steps) at once if it is possible. However `make` is smart enough to not try launch together steps that are dependent on each other.

But how to make single process make use of multi-core ?

2.1. Distributing program work flow onto multiple CPUs

Sequential code programs that are big enough get divided into functions and modules executed in chain of operations. Some of them must run exactly after the others, but for some the order does not matter.

This means that we can take pull them out and put them into separate threads. This is easiest way to cope computer games with multi-core: putting graphics rendering, audio generation, and game logic into separate threads, or in case of image, audio and video compression: putting input codec and output codec, or transformation steps into separate threads.

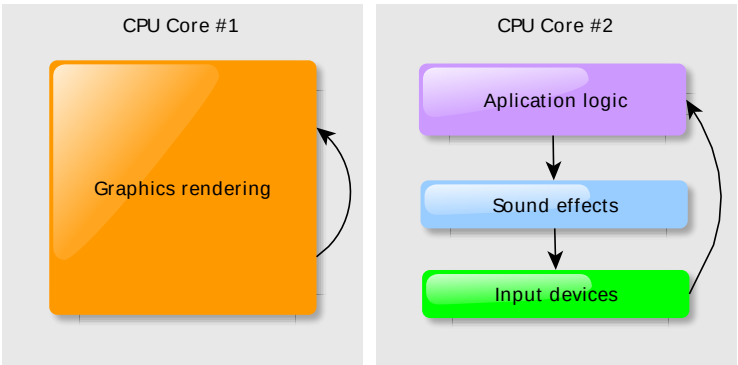

Figure 2. 2 CPU PC game application flow

Of course we cannot expect huge performance improvement, neither that it will make use of more processors than the modules running in separate threads, but this is good way to get some tens percent more out of the performance.

Our enthusiasm may be cooled down once we realize that for our PC game 95% of the CPU time is graphics rendering which cannot be really sliced into separate code blocks, so the other 5% running in the separate thread does not bring great improvement.

2.2. Splitting data and parallelizing data chunks processing

Another approach is to split the processed data into small chunks and let the each CPU core handle the chunk in its own separate thread. This is perfect solution for making use of old audio or image compression algorithms and many CPUs or cores. Audio and image data are easily splittable, and formats such as MP3 (audio) or JPEG2000 (image) fit to glue the output compressed chunks back together.

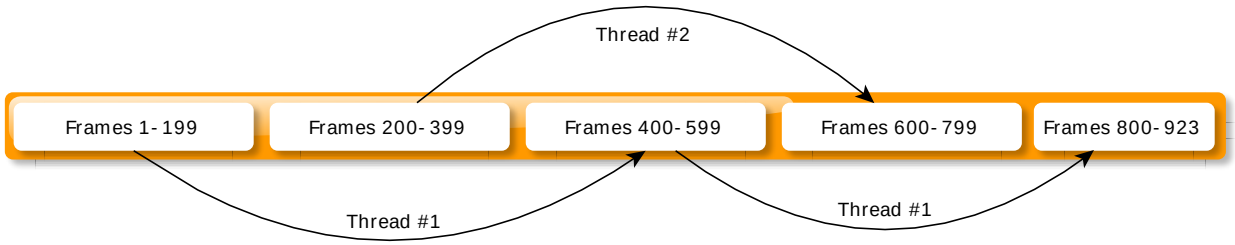

Figure 3. Splitting MP3 compression into chunks

As the result of such processing is usually written onto the disk, we may create separate file per each chunk and join them together after work is over. Of course we may suffer some time penalty from file join, but we can imagine that writing an extension for Linux ext3 kernel driver that will join data blocks of several files into one single file with `O(1)` should be an easy task.

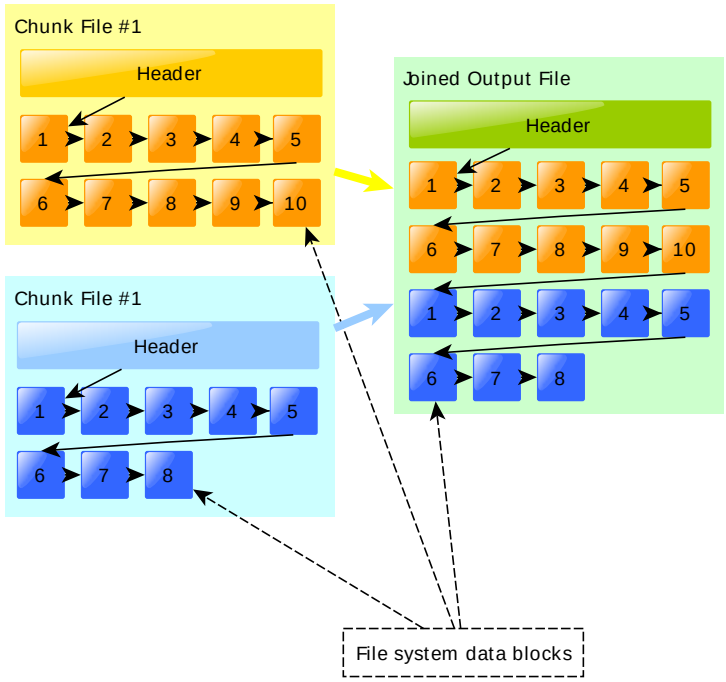

Figure 4. Joining files using file blocks list join

3. Looking into the future

Shifting to parallel computing does not need to be harmful or rapid. We can make gentle transition into parallel world step by step as the number of new processor cores grow. Today we know how to deal with 2-8 cores, but certainly not what to do with more than 100 cores.

Now we think with processes and threads, but once we hit level of hundreds or thousands of cores we may need to change our perception of computer algorithms. This may be a great chance for functional languages such as Erlang[6] or Caml (OCaml)[7] that just define functional relations of our program producing kind of execution relation graph that can be deployed onto the grid of the CPU cores. Soon the computers will resemble human brains with cores being equivalent of neuron.

There are also transactional memory, object cache storage, dynamically typed languages ideas on the horizon. All of those are gaining their momentum every day will be revised by the future soon.

Finally nVidia CUDA[8] effort must be mentioned here. As recent GPU (Graphics Processing Unit) chips can be considered as very multi-core processors, why not to make an use of them for regular computing. This is a concept behind CUDA (Compute Unified Device Architecture). It brings the future today, allowing launching parts of the regular C code on the GPU chip. Since recent GPUs have greater processing power that any regular CPU, GPU enabled programs can really perform fast. Of course the price is a need to rewrite your program to possibly state-less or small state, multi-pass parallel architecture, but the bounty is great.

References

Herb Sutter. *The Free Lunch Is Over*. Dr. Dobb's Journal. 2005.

Randall Hyde. *Write Great Code, Volume 1 & 2*. No Starch Press. 2007.

Daniel P. Bovet. Marco Cesati. *Understanding the Linux Kernel*. O'Reilly. 2001.

Vlad Pirogov. *The Assembly Programming Master Book*. A-LIST, LLC. 2005.

Joe Armstrong. *Programming Erlang*. The Pragmatic Bookshelf. 2007.

Michael J. Dickheiser. *C++ For Game Programmers, Second edition*. Charles River Media. 2007.

NVIDIA CUDA Compute Unified Device Architecture, Programming Guide. NVidia Corp.. 2008.

[6] Erlang is a programming language designed at the Ericsson Computer Science Laboratory. `http://www.erlang.org/`

[7] Caml is a general-purpose programming language, designed with program safety and reliability in mind. The Objective Caml system is the main implementation of the Caml language. `http://caml.inria.fr/`

[8] NVIDIA CUDA™ technology is the world's only C language environment that enables programmers and developers to write software to solve complex computational problems in a fraction of the time by tapping into the many-core parallel processing power of GPUs. `http://www.nvidia.com/object/cuda_home.html`