Available online at www.sciencedirect.com
ScienceDirect
Computers and Mathematics with Applications 53 (2007) 880–885
An International Journal
computers & mathematics
with applications
www.elsevier.com/locate/camwa

On the importance of cache tuning in a cache-aware algorithm: A case study

Péter Burcsi*, Attila Kovács

Department of Computer Algebra, Eötvös Loránd University, Budapest, Hungary

Received 20 June 2005; accepted 22 January 2007

Abstract

In the present paper we describe and analyse a sieving algorithm for determining prime numbers. This external memory algorithm contains several parameters which are related to the sizes of the levels in the memory hierarchy. We examine how we should choose the values of these parameters in order to obtain an optimal running time. We compare the running times obtained by varying the parameters. We conclude that in this specific problem fine tuning pays off as we got a speed-up of almost 40%.

Keywords: Cache-aware algorithm; Cache-tuning; Computational number theory; Sieving algorithm

1. Introduction

1.1. External memory algorithms

In modern computers, data are organised in a hierarchy of memories. The levels of memory differ in size and speed. This structure called for a new paradigm within computer science which is the theory of external memory algorithms. One of the first appearances of this theory can be found in [1]. For an introduction, we refer to the excellent survey by Vitter [2]. External memory algorithms are algorithms that deal with data that don't fit in the memory. They can be modelled in several ways, one of them being the two-level hierarchy model [3] and another is the multi-level model [4]. In this latter model there are several levels of memory each containing data. The closer a level is to the CPU, the faster but smaller it is. At the bottom of this hierarchy there is a level that is presumed to be infinitely large but very slow (in practice this can be thought of as hard disk). In this theory algorithms are not only compared by their consumption of CPU time but also the I/O operations they perform. The main goal in practice is finding algorithms that run as fast as possible. A good external memory algorithm is usually one that is aware of the underlying hierarchy and uses its knowledge about memory sizes and speeds so as to move data in an organised way.

1.2. Cache-aware and cache-oblivious algorithms

For some problems, there exist algorithms that know nothing about the memory hierarchy (apart from its existence) and yet perform in an asymptotically optimal way. An algorithm that cannot use the memory sizes in its decisions

* Corresponding author.
E-mail addresses: bupe@compalg.inf.elte.hu (P. Burcsi), attila@compalg.inf.elte.hu (A. Kovács).

0898-1221/$ - see front matter
doi:10.1016/j.camwa.2007.01.004

is called cache-oblivious [5]. We distinguish between cache-aware and cache-oblivious algorithms. One expects that cache-aware algorithms should run at least as well as cache-oblivious ones. In practice, however, the optimal values of the parameters needed by a cache-aware algorithm might not be straightforward from the actual physical sizes of memory levels: a bad parameter may strongly influence the running time. In what follows we fine tune a cache-aware algorithm to its optimal speed and then examine how a bad parameter influences its running time.

1.3. The test problem

The algorithm we considered had been invented for a mathematical problem in computational number theory. We have to determine which numbers are prime in a very large interval of numbers. The numbers are represented by an array of bits and the composite numbers are "sieved out". For this we must read and write most of the memory blocks containing the array several times. The algorithm was implemented mainly in assembly and C and was run on a multi-processor supercomputer (we give the details of our running environment later). The levels of memory were: a fast level 1 cache, a level 2 cache, main memory and disk. As speed was a crucial point for us, we did not let the data "flow out" of main memory in order to avoid disk I/O operations. The two parameters we considered were the sizes of the two caches. However, the algorithm did not run optimally using the exact physical sizes as parameters.

1.4. Tuning the program

The tuning was done by measuring running times and the number of memory operations in test cases. We used the built-in hardware performance counters of the computer to obtain data. We show how tuning improved the performance and how the analysis of tuning data resulted in a slight change of the original algorithm which also led to a speed-up. We conclude that in our case, where the program is meant to run for several weeks (maybe months) it pays off to extensively tune the algorithm.

2. Description of the test program

The program we tested was written for number theoretic investigations such as calculating statistics of gaps of consecutive prime numbers, calculating an approximation of Brun's constant, finding so-called curious prime combinations etc. All this comes down to determining all prime numbers in an interval. The part of the program that we analysed in detail is a sieving algorithm that has an interval as input and has the prime numbers of this interval as output. Hitherto sieving is the best known method for finding all prime numbers in a large interval (obviously, one has better algorithms when the primality of a single number is to be decided). The implementation of the sieve is mainly due to Járai, who used this technique together with his coauthors in their record prime searches like [6], or more recently [7].

2.1. The sieving algorithm

Since all prime numbers greater than 2 are odd, we can restrict our attention to odd numbers. If the input is an interval of length l, we suppose that the odd numbers in the interval are represented by an array of bits of length l. Initially all the bits in the array are set to 1. In the end, only the positions corresponding to prime numbers remain 1s, while the other bits are 0s. In our case the starting and endpoint of the interval is always between 0 and 2^{54}, this latter number being the upper bound for our calculations. Since all odd composite numbers in this range are multiples of some odd prime smaller than 2^{27} we can grab the basic idea of the sieve in the following pseudocode:

```
SIEVE_INT(START, LENGTH, LBOUND, UBOUND)
1   for all prime p with LBOUND < p < UBOUND
2       find first odd number after START that's a multiple of p
3       set every pth bit to 0, up to START + LENGTH
```

Here we put LBOUND $= 0$ (we only use this variable for the sake of the following piece of pseudocode) UBOUND $= 2^{27}$, START is the first number (odd) in the interval, and LENGTH is the length of the array. The prime numbers smaller than 2^{27} are calculated in advance and stored in a specific data structure. This data structure makes

easier the first step in line 2 in the above loop (if the function is called several times), since together with the prime, the residue of the first element of the interval is stored, too. For place saving the gaps between consecutive primes rather than the primes themselves are stored.

The above pseudocode is deliberately simplistic, not taking into account the memory structure. Therefore one can think of a program running in interleaved loops:

```
SIEVE(START, LENGTH, BOUND)
1  find B1, B2 with B1 < B2 < BOUND in some clever way
2   for n = 0 to LENGTH/B2
3      for m = 0 to B2/B1
4          SIEVE_INT(START + n*B2 + m*B1, B1, 0, B1)
5      SIEVE_INT(START + n*B2, B2, B1, B2)
6   SIEVE_INT(START, LENGTH, B2, BOUND)
```

Again START, LENGTH are as above, and let $\text{BOUND} = 2^{27}$. Here's a bit of explanation on how this should be read. We divide the sieving primes into three groups: small, medium and large ones. B_1 and B_2 are the sizes of the first two levels of memory in the hierarchy (level 1 and level 2 caches). Multiples of small primes are supposed to be close enough in the array so that a whole block of the array fitting in level 1 cache contains several multiples of small primes. Once this block is present in the level 1 cache, it is worth sieving with all small primes. This is the case with medium primes and level 2 cache as well. Large primes are thought of as ones that most probably create cache read/write misses anyway.

The sieving is done as follows: a block of the array is read into level 1 cache and sieving is performed by all small primes, one after the other. Then an adjacent block is read and sieved and so on. We know that recently used memory lines are not likely to get out of a level of memory until it is full. So we gather small blocks of the array together until they give a "medium" block, just fitting in the level 2 cache. We then sieve by medium primes, one by one. After that another medium block is sieved by small and medium primes and so on. We finish by large primes.

From the above method we hope to get as few cache misses at both levels of the cache as possible. It turned out however that choosing the optimal parameters for our embedded cycles is not obvious at all. We analyse this in detail later.

2.2. The implementation

The pseudocode given above is just a very simplified version of the implemented algorithm but it shows its cache-awareness. The routine SIEVE_INT was programmed in assembly by A. Járai in a fine way, naturally with a lot of enhancements. The calling routine SIEVE was written in C. When called, the program reads the parameters START, LENGTH, B_1, B_2. We experimented with various settings to obtain the fastest possible version.

3. The running environment

We give a brief description of the computer we used, concentrating on the memory hierarchy and on the built-in hardware performance counters.

3.1. The architecture

The computer we used is a Sun supercomputer of the Hungarian Academy of Sciences. It consists of three servers: two Sun Fire 15000 and a Sun Fire 480R machine. One of the 15000s has 64 US-III+ 1050 MHz processors, the other has 64 US-III 1200 MHz processors. The 480R has 4 US-III 900 MHz processors. The quantity of memory is 128, 132 and 8 GB, respectively. The detailed description of the UltraSPARC III Cu processor can be found in [8] or in [9]. The complex memory hierarchy contains lots of caches only two of which (the data caches) are of special interest to us. The level 1 data-cache is a 64 kB, 4-way associative cache, while the level 2 cache is 8 MB in size, and doubly associative.

3.2. The performance counters

The IIICu processor comes with built-in hardware performance counters [10]. We used a function library called CPC to tune our program. This C-library allowed us to use the performance counters. These counters can show how much time was spent by executing the code and how much of it was spent in specific functions. This can be measured in wall time or processor cycles. One can also break down the number of cycles during the execution into various types: cycles lost due to data cache read misses, instruction cache read misses, level 2 cache read misses, write misses or cycles spent waiting for integer/floating point operations to finish etc.

From the nature of our program we expected that most of the execution time is due to memory accesses. Execution data show that this was the case.

4. The tuning of the algorithm

First we determine in how large portions the sieve should work. Then we break down the running time into different operations so as to be able to see what parts of the program are the most expensive. After that we perform the real fine tuning.

4.1. Basic settings

We needed to calculate with all the prime numbers up to 2^{54}, but fortunately, we could only do this with smaller portions at a time. To get the prime numbers in an interval, we have to initialise our routines, so it is worth sieving a larger interval. But if it is too large, we may spend much time in the memory or even worse, on the disk. So we have to trade off between initial work and memory access.

We found that the wall-clock time of the program can differ by a factor of 3 or more between different calls therefore performance counters are used. We measured the running time of the program (and the inner loops, separately) in units of CPU cycles. We compared the cycles needed for various interval lengths. Here are a few measurement data for various interval lengths.

Interval length	CPU cycles	Speed (words/1000 cycles)
16 777 216	43 846 830 889	0.38
12 582 912	31 039 008 164	0.41
8 388 608	7 014 107 795	1.20
4 194 304	3 445 143 175	1.22
2 047 152	2 026 624 528	1.01
1 048 576	1 149 872 061	0.91
524 288	802 613 248	0.65

Here the interval length is given in 64 bit computer words. It is easy to observe the speed-up as the interval gets longer and an eventual slow-down as we reach a limit and hard disk I/O operations appear. We chose the near optimal value of 32 MB as the size of the portions. The next task was to break down the time consumption into different types of operations.

4.2. Finding the expensive operations

We use the hardware counters to find which operations are expensive. We break down the CPU cycles into several sub-categories. We give the results in the following table, omitting the counters that are not significant:

Operation	Cycles due to this operation (%)
Level 1 data cache read misses	14.0
Level 2 cache read misses	64.0
Waiting for integer operations to finish	0.3
Cache write misses	6.1

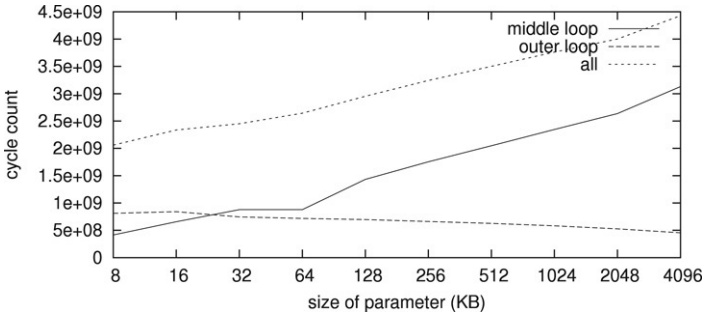

Fig. 1. The running time as function of parameter B_2.

In this table we can see how much of the overall running time was spent doing various operations. These numbers don't sum up to 100%. The remaining time is spent doing effective work (CPU work). We can see that accessing data not present in caches is the most expensive operation. Furthermore as the access speed of main memory is much lower than that of level 2 cache, the goal is to organise data so as to minimise the number of cache misses. In the description of our algorithm two parameters B_1 and B_2 were left undetermined. Now we can tune our algorithm by changing the values of these two parameters.

4.3. Playing with the parameters

One can reduce the number of I/O operations by exploiting spatial and temporal locality in data access. That is, once a block of memory is accessed and transferred to the cache, we should keep it there as long as possible.

Let's have a look at what happens if we change B_2. If we increase it some large primes become medium ones. So instead of sieving with them in the outer loop, we do it in the middle loop. This means that they are likely to generate fewer level 2 cache misses since once a block of memory is in cache 2, all medium primes benefit from its presence. But we have to be careful and we should keep B_2 below the size of the level 2 cache. The same is true for B_1 with the innermost loop. We expect the ideal values to be around the sizes of the caches. Here's what we got in the tests. We found that changing B_1 doesn't influence the running time very much. So we focus on B_2. The measured running times can be seen in Fig. 1.

As we expected, we spend less time in the outer loop and more in the middle loop as B_2 gets larger. Surprisingly, the overall time increases, too. A deeper insight into the middle loop has shown that the expensive cache misses are not equally distributed among different iterations of the loop. There are a few iterations with lots of cache misses then many iterations without any cache misses at all, then again, iterations with cache misses etc. Furthermore, the distance between these "clusters" of expensive iterations was always a power of 2. This distance is inversely proportional to B_2. Their constant product is the size of the level 2 cache.

What is the explanation of this strange behaviour? It seems that some positions in the cache are "bad" and this can be the result of something else occupying those positions. This turned out to be the data structure storing the pre-calculated sieving prime numbers. We found that the low associativity of the cache is responsible for this concurrence between two sets of data. In the bad positions, whenever a new prime was read the block of memory actually being sieved was put aside because of a cache conflict, then during sieving the block containing the next prime is dismissed and so on. So it is only worth sieving by a prime if enough work is done within a single block of size B_2.

4.4. The modified algorithm

We modified the initial algorithm as follows:

```
SIEVE(START, LENGTH, BOUND)
1  find B1, B2 k, l with B1 < B2 < BOUND in some clever way
2   for n = 0 to LENGTH/B2
3     for m = 0 to B2/B1
4        SIEVE_INT(START + n*B2 + m*B1, B1, 0, B1/k)
5     SIEVE_INT(START + n*B2, B2, B1/k, B2/l)
6  SIEVE_INT(START, LENGTH, B2/l, BOUND)
```

The slight modifications in lines 4–6 have as a result that critical sieving primes are left out of the inner and middle loops. We experimented with several values of k and l and choosing both of them around 10 turned out to be right. After some more tests we found that with the modified algorithm B_1 and B_2 are optimal if they are just below the cache sizes. Altogether we reached a speed-up of 40% compared to the original version.

5. Summary

In our case study we have shown a simple algorithm which had been designed to run effectively on a computer with several levels of memory. Trying to tune the parameters for optimal performance gave unexpected results. This shows that porting an algorithm to a specific platform requires a thorough understanding of the hardware which, in our case is especially true for the memory hierarchy and management. We found that a slight modification in the original algorithm could give a speed-up resulting in a running time reduced by 40%.

We conclude that tuning a cache-aware algorithm remains an important issue in algorithm engineering. Even when theoretical reasoning predicts optimal performance, parameters and eventually the algorithm should sometimes be modified.

Acknowledgement

The research was partly supported by Ericsson-ELTE CN Laboratory, and for the second author by OTKA-T043657 and Bolyai Stipendium.

References

[1] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Communications of the ACM 31 (9) (1988) 1116–1127.
[2] J.S. Vitter, External memory algorithms and data structures: Dealing with massive data, ACM Computing Surveys 33 (2) (2001) 209–271.
[3] J.S. Vitter, E.A.M. Shriver, Algorithms for parallel memory I: Two-level memories, Algorithmica 12 (2–3) (1994) 110–147.
[4] J.S. Vitter, E.A.M. Shriver, Algorithms for parallel memory II: Hierarchical multilevel memories, Algorithmica 12 (2–3) (1994) 148–169.
[5] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: FOCS '99 Proceedings of the 40th Annual Symposium on Foundations of Computer Science, 1999, p. 285.
[6] K.-H. Indlekofer, A. Járai, Largest known twin primes and Sophie Germain primes, Math. Comp. 68 (1999) 1317–1324.
[7] T. Csajbók, G. Farkas, A. Járai, Z. Járai, J. Kasza, Report on the largest known Sophie Germain and twin primes, Ann. Univ. Sci. Budapest. Sect. Comput. 25 (2005) 181–182.
[8] Sun Microsystems, UltraSPARC-IIICu User's Manual, Version V1.0, May 2002. http://www.sun.com/processors/manuals.
[9] R. Van der Paas, Memory Hierarchy in Cache-based Systems, Edition November 2002. http://www.sun.com/blueprints.
[10] D. Gove, Using UltraSPARC-IIICu Performance Counters to Improve Application Performance. http://developers.sun.com.