FIRA - A novel method for benchmarking the cache hierarchy.

Varad R. Deshmukh
College Of Engineering Pune
Shivajinagar, Pune, 411005
deshmukhvr07.comp@coep.ac.in

Nishchay S. Mhatre
College Of Engineering Pune
Shivajinagar, Pune, 411005
mhatrens07.it@coep.ac.in

Shrirang K. Karandikar
Computational Research Laboratories
TACO House, Damle Path
Pune, 411004
shrirang@crlindia.com

ABSTRACT

CPU Micro-architecture has a significant impact on performance and hence benchmarking micro-architectural performance is of utility to the High-Performance Computing industry. In lieu of conventional benchmarks, benchmarks that reflect the performance of micro-architectural features, independent of the application profile, are required. Memory hierarchy is an important part of the micro-architecture, having a major impact on performance. In order to make an application independent characterisation of the memory hierarchy, it is necessary to measure its important performance parameters, namely the penalty for a cache miss in different cache levels. The conventional program used for analysing the memory system does not provide direct and cycle-accurate measurements for all levels of the hierarchy. This work presents a novel method of benchmarking the cache. It uses a unique access pattern to generate a deterministic number of cache misses in every level and measures the access time. Using this method the access time for each cache level can be directly obtained. We describe the method in detail herein and discuss the results obtained for different micro-architecture variants. We check the results for consistency using statistics from numerous runs. We also check the results against those provided by the traditional method. Finally, we compare the estimated number of cache misses caused by our deterministic method with the actual number of misses, as measured by hardware performance counters.

Categories and Subject Descriptors

A.3 [**High Performance Computing**]: Architecture

General Terms

Performance,Measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Compute 2012 January 23–24, 2012, Pune, Maharashtra, India

Copyright 2012 ACM 978-1-4503-1440-4/12/1 ...$10.00.

Keywords

micro-architecture,benchmarking, cache

1. MOTIVATION

The technique presented in this work was developed as part of a comprehensive suite of micro-architecture benchmarks. Therefore, we first briefly examine the motivation behind micro-architecture benchmarking in general, benchmarking cache accesses in particular and the design of this method. We also look at the previously implemented techniques of timing cache accesses, how this method is different from the existing methods and what advantages it presents.

1.1 A Micro-Architecture benchmarking suite

Implementation of the instruction set architecture (ISA) of the computer – the micro-architecture, is a key factor influencing performance growth in the CPU industry. The performance differences between micro-architecture variants are barely noticeable to the average user, especially if the products run at comparable clock frequencies. However, the high performance computing (HPC) industry uses thousands of microprocessors to build clusters that are required to provide a cumulative throughput of the order of millions of instructions per nano-second. In the face of growing demands on performance, nano-second level improvements in the performance of processors do make a significant difference. In order to make informed and correct decisions for purchase of high performance CPU's, evaluation and comparison of micro-architectures is of great importance and utility to the HPC industry.

The traditional approach to processor benchmarking, involving the running of high-level application programs on the machines and comparing the total run-times, fails to relate performance to the micro-architectural characteristics, prominent benchmarks using this approach being the SPEC, Stream or LINPACK suites. It is inadequate for making comparisons along particular features of micro-architecture as well as for exposing similarities and differences between processors of comparable clock frequencies.

While hardware should be evaluated on the basis of the performance of its constituent components, the results provided by the current approach seem to be inexorably tied to the execution profile of the benchmark programs. A survey of existing work in micro-architecture performance evaluation reveals a fragmentation in terms of purpose of the works, hardware features studied and methodology. Micro-architecture is inherently invisible to the programmer and

measuring its characteristics using software is the most difficult part of micro-architecture benchmarking, requiring the use of unconventional benchmark programs.

The need was felt for a comprehensive suite of micro-architecture benchmarks and a method of analysis which enables us to compare processors on the merits of the hardware itself. We created such a benchmark suite and technique. Our approach was aimed at systematically addressing all the shortcomings of the existing approach by filling in the gaps and creating new techniques where required. As we shall see, such a technique was required for the memory hierarchy.

1.2 Benchmarking the cache hierarchy

In every computer program that performs some useful computation, there is at least one datum input from memory or one output to memory, or both. As per a study by Hennessy and Patterson [1] using the SPECint92 suite, 42 % of instructions are data memory accesses and memory performance lags processor performance by the order of 10^4 as of 2010. As such, memory access time can still be termed the biggest hindrance to performance.

Cache memory uses the principle of spatial and temporal locality of data accesses to significantly reduce the average memory access time. It has been shown in [1] that an ideal cache can reduce the average memory access time by an order of 10^2.

However, in practice a cache is often missed, depending on the locality of references displayed by the program. It can be shown that if the count of cycles per instruction (CPI) is one, and data accesses consist of 50 % of the instructions, and cache miss penalty is just 1 clock cycle, a program bringing about a miss rate of just 0.02 will cause the processor to run 1.5 times slower than the one with the ideal cache. This 'slowdown' grows in proportion to the cache miss penalty.

One of the main goals of our micro-architecture benchmark suite is to be comprehensive in the coverage of features that best represent CPU performance. Evidently, hierarchical memory (cache) is a feature that has a major influence.

We also include instruction cache, translation look-aside buffers, and RAM performance in the list of these metrics. However, we focus on the data-cache hierarchy for this work.

1.3 Previous work and the traditional method

Among the works that characterise micro-architecture for performance evaluation, we observe that data cache benchmarks are by far the most prolific, used in almost all studies including [2] and [3]. Available commercial or Open Source benchmark packages like STREAM [4], Ramspeed [5], 7-bench [6] also focus chiefly on the data cache.

Karkhanis et al. [7] present the different aspects of data cache behaviour using simulation. No work involving performance evaluation of the instruction cache is encountered.

In data cache benchmarking most of them use the same technique, introduced by Saavedra-Barrera [8]. We discuss this in brief here, to get a idea of its shortcomings for our purpose.

A large array is allocated to the benchmark program. This array is several times larger than the largest data cache. There is a loop, which accesses this array sequentially, with a particular stride. A set of timer functions is used to time the execution of this loop. This is repeated multiple times. This experiment is carried out on array sizes from 4 KB to 64 MB and varying the stride from 1 to half the array size, for each array size.

The code for this benchmark can be stated as :

```
for(array_size=4K; array_size < 64M; array_size *=2)
    for(stride=1; stride < array_size/2; stride*=2)
        for(i=0; i < array_size; i+=stride)
            k=array[i];
```

When we plot the timings for the array access against stride sizes, for increasing array sizes, we see a graph like the one shown in the figure 1. The timings represent the average memory access time (AMAT) in nanoseconds. This particular figure represents the results for the Pentium-4 (Prescott) machine, running at 2.8 GHz.

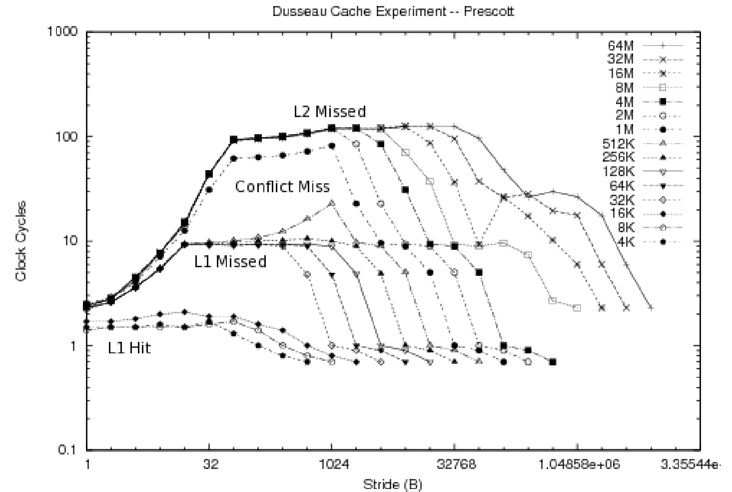

Figure 1: Results of the traditional memory benchmark

We know that the Prescott has two levels of cache, the first (L1) being 16 KB in size and the second (L2) being 1 MB. We also know that both have cache lines 64 bytes in size.

In figure 1, we can observe that for strides smaller than 16 (that is 64 B, since each element is 4 B) and arrays smaller than 16 K, the AMAT is very low, at around 1 nanosecond. This may be attributed to the fact that such accesses hit in the L1 cache. Thus, the AMAT for this region represents the L1 cache hit time.

Once the stride is greater than 16, we see that even for arrays fitting into L1, the AMAT increases. This is due to the fact that consecutive accesses now fall in different cache lines, causing cache misses. This inflection point corresponds to 32 (128 B), double the size of our cache line. After this point the AMAT is always the weighted sum of hit time and miss penalty for the L1 cache.

For arrays greater than the size of L1, the AMAT is distinctly higher right from the smallest strides, owing to the increased number of cache misses in the form of 'capacity misses'. This is represented by a distinct region of the plot

– arrays from 32 K to 512 K – which fit the L2 cache. In this region and above, conflict misses also start playing a part – as the stride approaches 1024, (4096 B) which is the size of the set (Prescott L2 cache being 8 way associative), the number of accesses in the same set increases, causing 'conflict misses'. This reaches its peak at 1024 stride.

For array size equal to the L2 cache (1 MB), we see a separate curve, not falling in any of the major regions. This may be due to the additional cost of TLB misses. For this curve and the curves above, the AMAT is the weighted sum of penalties from misses in the L1, L2 and TLB and the RAM access time.

Certain observations about this method itself are in order. Firstly, the timings have to be calculated by an analysis of the output data like the one elaborated above. The output itself is not the measured value.

An advantage of this method is that most of the inflexion points correspond to the sizes of cache features. Therefore it is extremely useful for dynamically knowing the cache characteristics when they are unknown, without using any assembly coding. The 'cpuid' instruction requires assembly coding and its output is different for different x86 vendors.

Except in the first 'region' (stride smaller than cache line size, array smaller than L1), all the timings are weighted sums of the cache access times. Unless we know the exact fraction of accesses that hit or miss a particular level, we cannot directly obtain the cache access timings. It is not possible to know before-hand the exact number of cache misses that this program causes; we can only estimate qualitatively.

The program does not account for the 'hardware prefetchers' that are present in the current CPU's, which detect a constant access stride and prefetch cache lines into the L2 cache. So qualitative estimates of the number of misses in the second region can actually be mistaken.

However, for our benchmarking approach, we require to know quantitatively, and as accurately as possible, the number of cache misses occurring, in order to measure the access times directly. We also desire to account for newer features like the prefetchers. Therefore, we developed the new method.

1.4 Benchmark development approach

We present our broad approach to the design of micro-architecture benchmarks. We first identify the main features of the micro-architecture, and map each feature to a set of execution 'events' associated with it. Every event has a quantitative measure associated with it. Micro-architecture benchmarks have to deterministically cause these events, and measure the respective quantities. To develop these, we have to determine the precise instruction sequences that give rise to the events of interest and generate code which represents this sequence. The performance of these code sections is thus, directly proportional to that of the micro-architectural feature under study. Such measurement characterises the hardware in terms of its own performance, helps us to meet our objective of workload independent benchmarking.

'Cache misses' are the events associated with the memory hierarchy. The penalty incurred in bringing data into the cache from main memory when a cache miss occurs, is the quantity of interest. This penalty depends on the access times for each level of the cache hierarchy. We therefore consider the access time for the primary (L1), secondary (L2) and tertiary (L3) cache levels as the essential metrics.

Our cache micro-benchmark has to cause a deterministic number of cache misses in each level of the hierarchy, and time the accesses accurately, to directly obtain the value of cache miss penalties.

This focus on determinism of events and direct measurement forms the core of our approach and is the main difference between our method and the popular method described in section 1.3. We will see in the next section how we map this approach to a benchmarking program.

2. FORWARD INITIALISATION-REVERSE ACCESS: (FIRA)

We now describe our method in detail. It is called Forward Initialisation - Reverse Access (FIRA). This method basically consists of initialising a large array, in the forward direction and then accessing it, in reverse, in a particular sequence, so as to cause a deterministic number of cache misses in each different level of the hierarchy. It is useful for benchmarking the cache for processors with an LRU replacement policy. Since most of the processors use this policy, we assume that this method works for a majority of machines. However, it is worthwhile to test how well this works with machines which have a different replacement policy. In fact some information about the replacement policy of the hardware could be found using this method.

We explain the FIRA method with a simplifying example of single level cache. An array 'A' is allocated from memory, whose size is greater than the size of this cache. [1]

We now access this array, by reading each element, sequentially from the first, to the last (Forward Initialisation). Each element is a word of 4 bytes. At the start, no part of the array is in the cache. If we access the array from the first index to the last we will be causing cache misses. But each element will be moved into the cache. However, since the size of the cache is smaller than our array, the entire array cannot be moved into the cache. After we have accessed part of the array equal in size to the cache, all further accesses will cause older elements to be evicted from the array and replaced by the newly accessed elements.

At the end of this initialisation, we will have a situation like the one depicted in Figure 2. If size of the array is $size_A$ and cache size is $size_C$, then the last $size_C$ elements of the array will be in cache. For the initialisation operation, we make sure that non-temporal writes are not used, so that the memory accessed is definitely read into the cache.

Now, if we access the same array, in reverse order, we are accessing elements already in the cache. As a result, the access time for each of the first $size_C$ elements from A[$size_A$] to A[$size_A - size_C$], will be equal to the cache access time. By timing this part of the program and dividing by the number of accesses – which we know as $size_C$ – we can exactly determine the cache access time. The figure 3 depicts the access pattern during the 'reverse access' phase.

It must be noted that during this phase, the accesses will have to take a stride equal to the cache line size in order to ensure that every access is a miss, since transfers between

[1]We require to know the size of the caches and of the cache lines before-hand. The values of cache sizes and other specifications can be obtained by the use of the 'cpuid' instruction available on Intel and AMD processors.

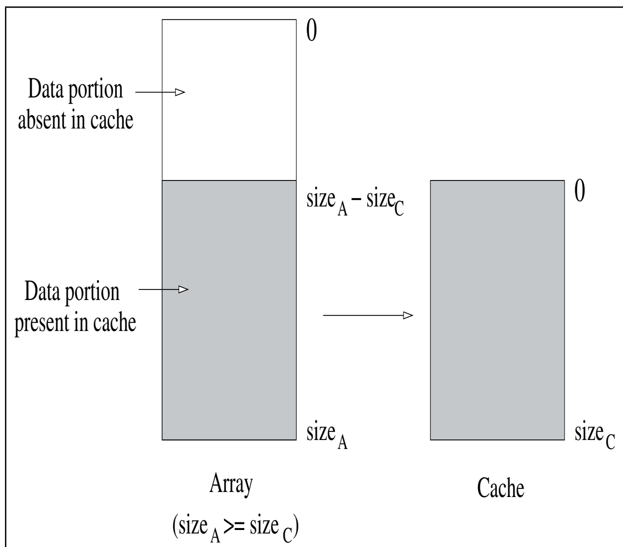

Figure 2: Mapping of Cache Levels to Array

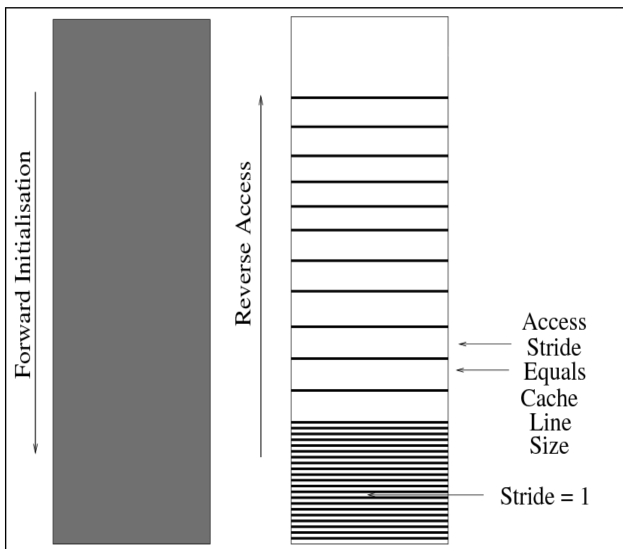

Figure 3: The access pattern used to isolate data cache levels

memory and cache take place in units of cache line – typically 64 bytes. Therefore the size of the remaining part, divided by 16 gives the number of accesses.

For all elements from A[$size_A - sizeC$] to A[0], all elements will have to be fetched from the main memory and then accessed from cache. Therefore, timing this phase of the program and dividing by the number of accesses – $(size_A - sizeC)/16$ – we can calculate the time for a cache miss penalty.

We can easily extend this to multi-level caches. Consider a two level system with L1 and L2 cache sizes denoted by $Size_{L1}$ and $Size_{L2}$ and a cache line size of 64 bytes (equivalent to 16 words). The procedure is the same, but with an array larger than the last level cache size. The forward initialisation will cause a layout similar to that shown in figure 2, but on multiple levels.

During reverse access we will first get as many L1 cache hits as the size of the L1 cache. These will be followed by L1 misses that are L2 hits, which should equal '$Size_{L2}$-$Size_{L1}$ / 16' in number. These will be followed by L2 misses, which are fulfilled from main memory or from L3 cache, if the processor has three levels. This multi-level version of FIRA is used as our data-cache benchmark in our micro-architecture benchmarking suite.

Certain processors use a hardware prefetcher, usually for the lower levels of cache. This prefetcher keeps track of access strides. If a constant stride is detected, then it prefetches blocks from the memory to avoid last level cache misses. In order to cause a fixed number of misses we need to neutralise the effect of the prefetcher. This is done using a randomly changing stride after when the last level cache miss phase starts. In this way we obtain the access times for L1, L2, L3 cache.

3. EXPERIMENTS FOR VALIDATION

This approach has not been tried before for cache benchmarking. Therefore, it needs to be validated. We first show the standalone results that the benchmark provides and then compare them with the results provided by the traditional method. We then check how our claim of 'determinism' holds by comparing the predicted number of cache misses with the actual number, measured using hardware performance counters.

In all these tests, a large number of instances of the benchmark were run to reduce the effect of noise.

3.1 Experiments with various micro-architectures.

We ran the FIRA micro-benchmark on a number of machines, each representing a different micro-architecture. The RAM latency was measured by a separate benchmark. The results are summarised in table 1.

As expected, the access time goes on increasing by an order of magnitude as the level goes from L1 to RAM.

Table 1: Memory hierarchy access times

Machine	Number of levels	L1 access time (clock cycles)	L2	L3	RAM	Frequency (GHz)
Prescott	2	1	17	-	358	2.80
Conroe	2	1	21	-	298	2.20
Clovertown	2	1	13	-	249	2.33
Clarkdale	3	2	4	19	360	2.93
Nehalem-EP	3	2	4	7	125	2.93

We observe that older processors (Prescott, Conroe and Clovertown) use one clock cycle for an access from L1 cache while the newer processors (Nehalem-EP and Clarkdale, both members of the Nehalem class of micro-architecture) use two. These observations agree with details of the Nehalem class of processors previously published by Intel [9], which state that the data access from a cache hit has been spread over two clock cycles in this class.

To check the consistency of these results, we collect results for one thousand runs of the benchmark on each machine and do a statistical analysis. [2] The statistics of the benchmark results (mean and standard deviations) for all the runs are shown in table 2. Time is reported in clock cycles.

Table 2: Statistics of the benchmark runs

Machine	L1 access time (mean)	L1 access time (SD)	L2 (mean)	L2 (SD)	L3 (mean)	L3 (SD)
Prescott	1.45	1.30	16.96	5.36	-	-
Conroe	1	0	21.05	2.86	-	-
Clovertown	1.04	0.40	13.33	1.10	-	-
Nehalem-EP	2	0	4.37	0.50	7.09	0.50
Arrandale	2	0	4.00	0.10	6.02	0.50

We observe a low standard deviation for all the data-sets but one (L1 hit for Prescott). Therefore, we have a high level of confidence in these results.

3.2 Validation using Performance Monitoring Counters

In order to validate the premise that the FIRA method can cause a deterministic number of cache misses in each level, we count the number of cache misses actually taking place, while the program is executing and compare it with the number of misses that we expect to cause. The program is the same. Instead of timing the code that causes the cache misses, we count the misses using the configurable performance monitoring counters that are found on Intel processors. Use of performance counters requires administrative rights for inserting the kernel module used as the driver for the counters. Due to lack of these rights on all the machines used in the study, only two machines among them could be used for this comparison.

We ran this test for those two machines, assuming that both implement LRU replacement.Table 3 shows the results for Conroe (Core micro-architecture) and Arrandale (Nehalem micro-architecture) for L1 and L2 misses.

There are two cache levels in Conroe and three in Arrandale. Thus, we cause only L1 misses in Conroe and both L1 and L2 misses in Arrandale. Since we are not using FIRA to measure the RAM access time, we do not consider last level misses here. The expected number of misses for L1, is the difference between sizes of L2 and L1, divided by the size of a cache line. Similarly for L2, the number is equal to the difference between sizes of L3 and L2 divided by cache line size.

We observe that our method is accurate to 98% for Conroe. However, for Arrandale, the accuracy for L1 is lower (81%) while that for the L2 cache is much lower, to a negligible degree. This may be attributed to the fact that the Nehalem micro-architecture family uses a policy different from LRU, causing this method to perform poorly. However, it is validated that this method is works for processors with LRU caches.

[2]The Clarkdale machine was unavailable for this analysis

Table 3: FIRA - expected and measured cache misses - Level 1

Machine	Expected number of misses	Measured	Percentage Accuracy
Conroe (L1)	32256	31618	98
Arrandale (L1)	3584	2912	81
Arrandale (L2)	45056	3975	8

3.3 Results from the Saavedra method

We used an implementation of the algorithm described in the first section, found in [1] for obtaining results of the traditional method, on all the machines listed in table 2. The table 4 summarises the results that we found from analysing the graphs obtained.

Table 4: Average Memory Access Times

Machine	Number of levels	AMAT (ns) (L1)	AMAT (L2)	AMAT (L3 / Last Level)
Clovertown	2	1.8	4.5	4.5
Conroe	2	2.2	9	9
Nehalem	3	1.8	2.2	8
Prescott	2	2	10	10

Comparing these latencies with the numbers from our experiments, we observe that the trends are similar but the numbers are not in exact agreement. This could be attributed to reasons we have discussed in section 1.3, namely, the results provided by the old method represent the average memory access time and not the exact cache access time for any particular level of cache. In contrast to the FIRA method, provides level-specific results.

Moreover, the running time of this benchmark is much higher than the FIRA benchmark. This makes it prone to system noise. We observed that the graphs cannot be exactly reproduced for successive runs of this on the same machine. The trends are similar, the inflexion points are the same, but the numerical values of the AMAT vary more than in case of FIRA.

4. CONCLUSIONS

We have developed a new method of measuring the cache access time in order to support a proposed new approach to CPU benchmarking using the micro-architectural characteristics. The new approach required direct measurement of access time for each cache level for processors with LRU caches.

This method, called FIRA, enables us to cause a deterministic number of cache misses in each level of the cache and determining the access time for that level. We tested the validity of this method using the statistics of multiple repetitions of this experiment and found that the results are accurate to clock cycle resolution, easily reproducible and highly consistent.

We also compared the number of cache misses expected to be caused by our benchmark to those actually caused, using hardware performance monitoring counters. We found that for a system with the LRU replacement policy, our estimate is highly accurate while it fails for a system with a non-LRU policy. Thus, more work is needed to extend this approach

to keep up with the latest techniques in memory system design.

5. REFERENCES

[1] J. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach, Fourth Edition*. Morgan Kaufmann Publishers, San Francisco, CA, 2007.

[2] J. Demmel. Single processor machines: Memory hierarchies and processor features; case study: Tuning matrix multiply.

[3] R. H. Arpaci-Dusseau et al. Empirical evaluation of the cray-t3d: A compiler perspective. In *ISCA'95*, pages 320–331, 1995.

[4] [Online]. Available: `http://www.cs.virginia.edu/stream/`

[5] [Online]. Available: `http://alasir.com/software/ramspeed/`

[6] [Online]. Available: `http://www.7-cpu.com/`

[7] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *In Workshop on Memory Performance Issues*, 2002.

[8] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, EECS Department, University of California, Berkeley, Feb 1992.

[9] [Online]. Available:`http://ark.intel.com/Product.aspx?id=37109\&processor=X5560\&spec-codes\%=SLBF4`