UPTEC IT 11 002
Examensarbete 30 hp
January 2011

Low Overhead Online Phase Predictor and Classifier

Andreas Sembrant

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Low Overhead Online Phase Predictor and Classifier

Andreas Sembrant

It is well known that programs exhibit time varying behavior. For example, some parts of the execution are memory bound while others are CPU bound. Periods of stable behavior are called program phases. Classifying the program behavior by the average over the whole execution can therefore be misleading, i.e., the program would appear to be neither CPU bound nor memory bound. As several important dynamic optimizations are done differently depending on the program behavior, it is important to keep track of what phase the program is currently executing and to predict what phase it will enter next.

In this master thesis we develop a general purpose online phase prediction and classification library. It keeps track of what phase the program is currently executing and predicts what phase the program will enter next. Our library is non-intrusive, i.e., the program behavior is not changed by the presence of the library, and transparent, i.e., it does not require the tracked application to be recompiled, and architecture-independent, i.e., the same phase will be detected regardless of the processor type. To keep the overhead at a minimum we use hardware performance counters to capture the required program statistics. Our evaluation shows that we can capture and classify program phase behavior with on average less then 1% overhead, and accurately predict which program phase the application will enter next.

Handledare: David Eklöv
Ämnesgranskare: Erik Hagersten
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT 11 002
Sponsor: SSF CoDeR-MP

Tryckt av: ITC

Sammanfattning

Swedish Summary

Det är välkänt att program uppvisar periodiska beteeden. Under en viss tidsperiod är programmet beräkningsbegränsat medan andra tidsperioder är minnesbegränsade. En *fas* kan definieras som en tidsperiod med ett likartat beteende. Det kan därför vara vilseledende att klassificera ett program som genomsnittet av hela programkörningen, dvs. programmet är varken beräkningsbegränsat eller minnesbegränsade. Flera olika optimeringar kan utnyttja fasbeteende, till exempel, schemaläggning av trådar [33, 22], kompilatoroptimeringar [3, 26] och strömförsörjning i processorer [16, 6, 13], m.m.

Mycket forskning har gjorts inom fasdetektering [32, 18, 20, 6, 16, 14, 29]. Vanligtvis görs det på följande sätt. Först delas programexekveringen upp i icke-överlappande intervaller. Under varje intervall mäts någon form av prestandabeteende, till exempel antalet funktionsanrop, *cache* missar eller *basic block* fördelning. Om två intervall är tillräckligt lika varandra klassificeras de till att befinna sig i samma fas.

En annan viktig del är att förutspå fasbeteende [32, 9, 29], vilken fas programmet kommer att befinna sig i härnäst. Ett flertal metoder har föreslagits [32, 20]. De undersöker programmets fasbeteende och gör en förutsägelse baserat på hur programmet betedde sig tidigare.

I det här examensarbetet utvecklar vi ett generellt bibliotek som kan klassificera och förutspå fasbeteende. Det följer vilken fas programmet befinner sig i och förutspår vilken fas som programmet kommer att byta till. Biblioteket har ingen direkt påverkan på programmet som övervakas, programmet behöver inte kompileras om, samt så är de detekterade faserna hårdvaruoberoende.

För att minimera exekveringtiden så använder vi hårdvaruräknare för att fånga *basic block*. Ett basic block är en sekvens instruktioner som alltid exekveras i samma ordning. En *basic block vektor* (BBV) är en endimensionell array där varje element talar om hur många gånger ett basic block har körts inom ett intervall. Två intervall klassificeras till att befinna sig i samma fas om Manhattan-avståndet mellan vektorerna är under ett tröskelvärde. För att ytterligare minimera exekveringstiden så använder vi oss av dynamiskt växande intervaller. Om programmet befinner sig i en stabil period så ökas storleken på intervallen och sampelhastigheten sänkes.

Vi utvärderar två olika metoder för att klassificera intervall. Först, en avståndsbaserad metod enligt ovan. Sedan, en sekvensiell version av K-Means [23], en vanlig algoritm för att gruppera data. För att förutspå faser utvärderar vi en så kallad *last value predictor* [32], den förutsäger att nästa intervall kommer att befinna sig i samma fas som intervallet innan, samt ett antal *Markov predictors* [32], som förutspår faser baserat på fashistorik.

Vår utvärdering visar att vi inte ökar exekveringtiden med mer än 1% i genomsnitt, och att vi kan med hög noggrannhet förutspå vilken fas programmet kommer att befinna sig i härnäst.

Contents

Acknowledgements

First of all, I would like to thank my supervisor David Eklov for getting me started on the project and all the helpful feedback along the way. I would also like to say thanks to my friend and colleague Peter Vestberg for technical feedback and interesting discussions on how to solve various problems. This thesis was funded by SSF CoDeR-MP.

1 Introduction

It is well known that programs exhibit time varying behavior [30]. A *program phase* is defined to be a period of execution during which the program exhibit a stable behavior. Program phases can reoccur several times during a program's execution. For example, the same function can be called from several call sites. There are several optimizations, such as thread scheduling [33, 22], compiler optimizations [3, 11, 26], simulation [31, 27], and power management [16, 6, 13], that have been shown to benefit greatly from considering the time varying behavior of programs.

Significant work have been done on program phase classification [32, 18, 20, 6, 16, 7, 10, 25, 14, 17]. It is typically done as follows: First, the application's execution is divided into non-overlapping fixed size intervals. For each interval various program metrics, such as function call frequencies, loop trip counts or basic block execution frequencies are collected. These metrics are used to compare the behavior of the intervals. If the behavior is similar enough the intervals are classified as belonging to the same program phase. Both offline [25, 5] and online [32, 9, 20, 24] approaches have been investigated.

Another important part of program phase analysis is phase prediction [32, 9, 29]. Phase prediction tries to predict what phase will be executed next. Several such methods have been proposed [32, 20]. They observe the program behavior and tries to predict the next phase based on previously seen patterns.

In this thesis we develop a general purpose online program phase classification and prediction library. It allows the user to attach to a running program and keep track what phase the application is currently executing, and query the library for what phase the program will enter next. This is done non-intrusively. i.e., the program behavior is not changed by the presence of the library, and transparently, i.e., it does not require the tracked application to be recompiled, and it is architecture-independent, i.e., the same phase will be detected regardless of the processor type. Importantly, the library does not significantly slow down the execution of the tracked application. This is achieved by using hardware performance counters to sample the basic block execution frequencies. Our evaluation shows that the execution time overhead is on average less than 2%.

We evaluate two different methods to classify execution interval. First, distance based classification [32], i.e., if the distance between two intervals is below a threshold they are classified into the same phase. Second, we used a sequential version [8] of the K-Means [23] clustering algorithm to cluster intervals, where each cluster is classified as a unique phase. For prediction, we evaluated last value prediction [32], which simply predicts that the next interval will belong

to the same phase as the current interval, and a set of Markov predictors [32], who predict the next phase based on the phase history.

In this thesis we make the following contributions:

- **Sampled Basic Block Vectors** (BBV) - Previous research [5] used hardware performance counters to sample *Extended Instruction Pointers Vectors* (EIPV). Lau et al. [18] have shown that larger control structures result in better accuracy. We sample branch instruction and can thus create basic block vectors [32].
- **Performance vs. Accuracy** - We investigate how the sample rate affect accuracy and performance, and we show that prediction is vulnerable to large sample periods while the classification accuracy remains relatively constant.
- **Dynamic Intervals** - We increase the size of the interval and lower the sample rate when we enter a stable period of execution in order to decrease the overhead. We examine the performance improvement and accuracy.
- **Sequential K-Means** - We evaluate how online K-Means clustering can be used to classify phases, and compare it with distance based classification.
- **Online Library** - We show a working solution how hardware counters can be used to classify and predict phases online, see Appendix A. Library users can then easily take advantage of program phase behavior for various optimizations.

The rest of this thesis is structured as follows. In Chapter 2, we discuss prior work and background on program phase behavior. In Chapter 3, we go into more detail and describe the methods used to capture, classify and predict program behavior. In Chapter 4, we evaluate the performance and accuracy. Finally we draw conclusions and discuss future work in Chapter 5.

2 Background

2.1 What is a phase?

It is well known that programs exhibit time varying behavior. A program phase is defined to be a period of execution during which the program exhibit a stable behavior. To classify a period of execution as belonging to a certain program phase, we first divide the program's execution into non-overlapping uniform intervals. By comparing a given performance metric measured during the intervals, we can compare the similarity of the intervals. If two intervals are similar enough they are considered to belong to the same program phase.

Figure 2.1 shows the execution of *gcc/166* for a set of performance metrics to better describe what a phase is and how program behavior varies over time. The x-axis shows time in number of executed instructions. Figure (l1d) show the hit rate in L1 data cache, (l2) show the hit rate in the unified L2 cache, (l3) show the hit rate in the L3 cache, (mpred-miss) show the number of branch miss-predictions, and (cpi) show cycles per instructions (CPI). We have labeled the four most prominent phases, A, B, C and D. A closer look at phase B shows that it occurs six times and that it has a very stable behavior for all performance metrics. If we look at the whole program behavior we discern a bigger pattern. The phase pattern $A-B-B-C-B-D$ reoccurs two times. See Appendix B for other SPEC applications.

2.2 Tracking phases by code

For general purpose phase detection it is important to find phases that remain stable for several performance metrics and are architecture independent. For example, if we were to use branch miss predictions (bpred-miss) to define phases, phase A and B could be considered the same phase. But looking at how the cache behaves we clearly see that they should be considered as two different phases. Phase A have a higher hit rate in the L1 data cache and phase B do not use the L3 cache at all. The program phase detection should not be affected by the execution environment, running the same program with the same input multiple times should produce the same result, i.e. we could have different results if we were to use cache misses to define phases due to cache sharing between programs.

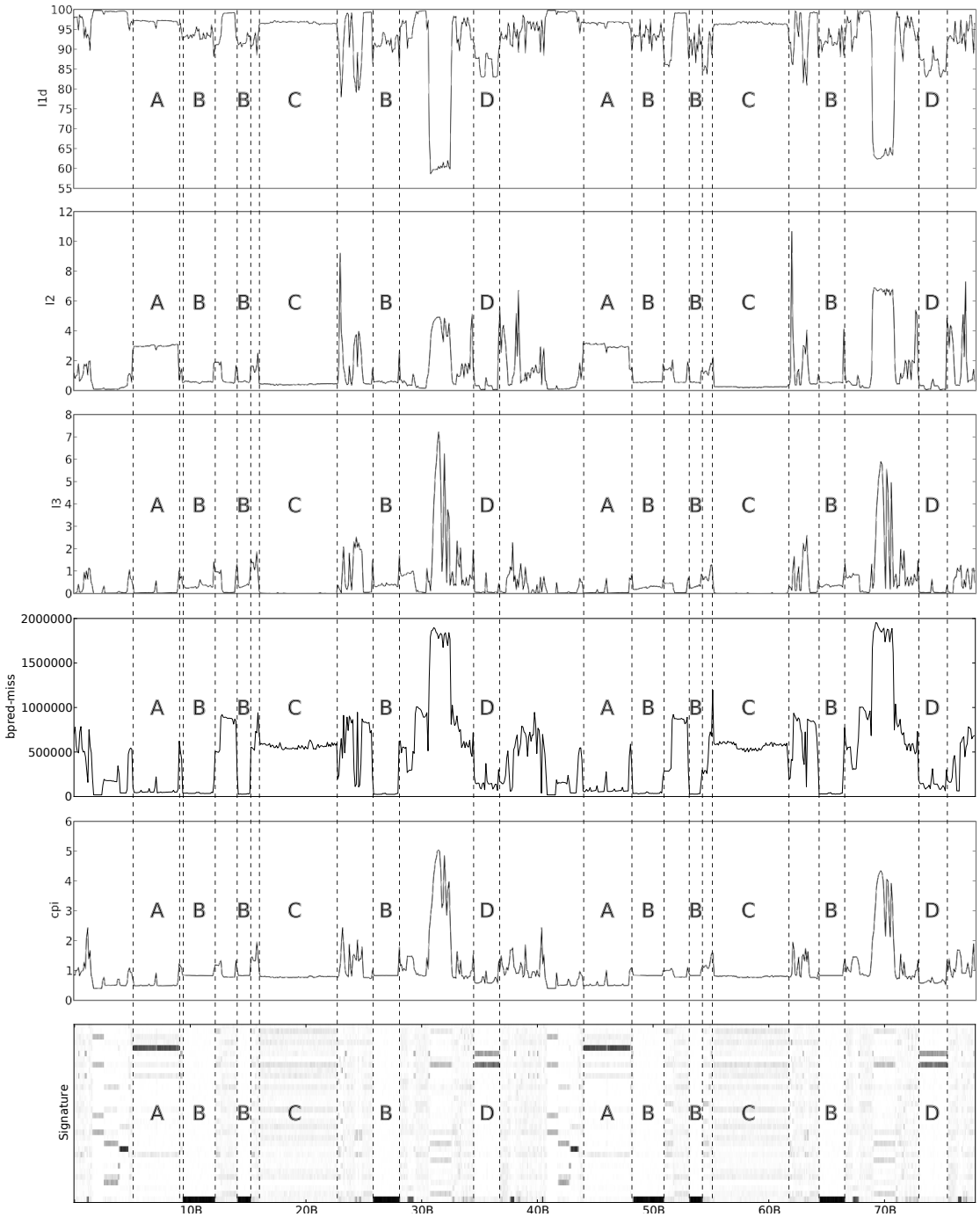


Figure 2.1: To show how program phase behavior changes over time, and how it affects different metrics, we have plotted the execution of *gcc/166*. The hit rate in L1 (l1d) data cache, the hit rate in the unified L2 (l2) cache, the hit rate in the L3 (l3) cache, the number of branch miss-predictions (mpred-miss), the average cycles per instructions (cpi) and the signature. Time is shown as number of billions instructions. Each metric was sampled at 100 million instructions intervals.

Lau et al. [18] have shown that most performance metrics are a result of what code was executed. They investigated possible ways to track phases using the code, how often certain control flow structures were executed, how often each operation code (opcode) was used and finally how often each register was used. They found that classifying phases by executed control flow structures, or register usage, produced very good results. Their evaluation was done on a RISC architecture. We speculate that the performance of register usage would most likely decrease for x86 and other CISC architectures with fewer registers. Handling all x86 instruction would also increase the complexity. We think that control flow structures are the best option.

2.3 Tracking phases by basic blocks

A basic block is a sequence of instructions that always execute together. It has a single entry point, meaning that no instructions inside the block are the destination of a jump instruction, and it has a single exit point, meaning that no instruction in the block is a jump instruction. All the instructions in a basic block are therefore executed the same number of times.

```
1 int fibonacci(int n) {
2   if (n == 0) return 1;
3   if (n == 1) return 1;
4   else        return fibonacci(n - 1) + fibonacci(n - 2);
5 }
```

Listing 2.1: Fibonacci

The Fibonacci function above is used to illustrate how basic blocks can be used. The function is first compiled with *gcc*, decompiled with *objdump*, and a *control flow graph* (CFG) is created by dividing the assembly code into basic blocks. Figure 2.2 shows eight basic block, solid arrows points from the exit of a basic block to the entry point of the succeeding basic block, dashed arrows show where the control flow falls through without a branch instruction. A function call has two outgoing arrows but the control flow will always return to the block below when the call returns. For example, the function call at E_{16} will first go to A_1. The control is then returned to F_{17} from H_{26}. As a result, basic blocks E, F, G will be executed the same number of times.

Dhodapkar and Smith [6, 7] used a bit vector with one bit per basic block to track the code. Each time a basic block was executed the corresponding bit is set. They could then classify two execution intervals as belonging to the same phase if they have similar bit vectors. One weakness with their approach is that two different intervals can be classified into the same phase. For example, consider the case with two intervals A and B, where interval A spends 90% in function F_1 and 10% in function F_2, and interval B spends 20% in function F_1 and 80% in function F_2. The two intervals would set the same bits and we classify them as belonging to the same phase but they execute the code in different ways.

To solve this problem Sherwood et al. [32] associate a counter with each basic block. Each time a basic block is executed its counter is incremented by one. They could now distinguish interval A from B, and classify them into different phases. Instead of keeping track of all basic blocks, they used a vector with a limited number of counters called a *basic block vector* (BBV). When a basic block is executed, the address of the block is hashed and used to index into the

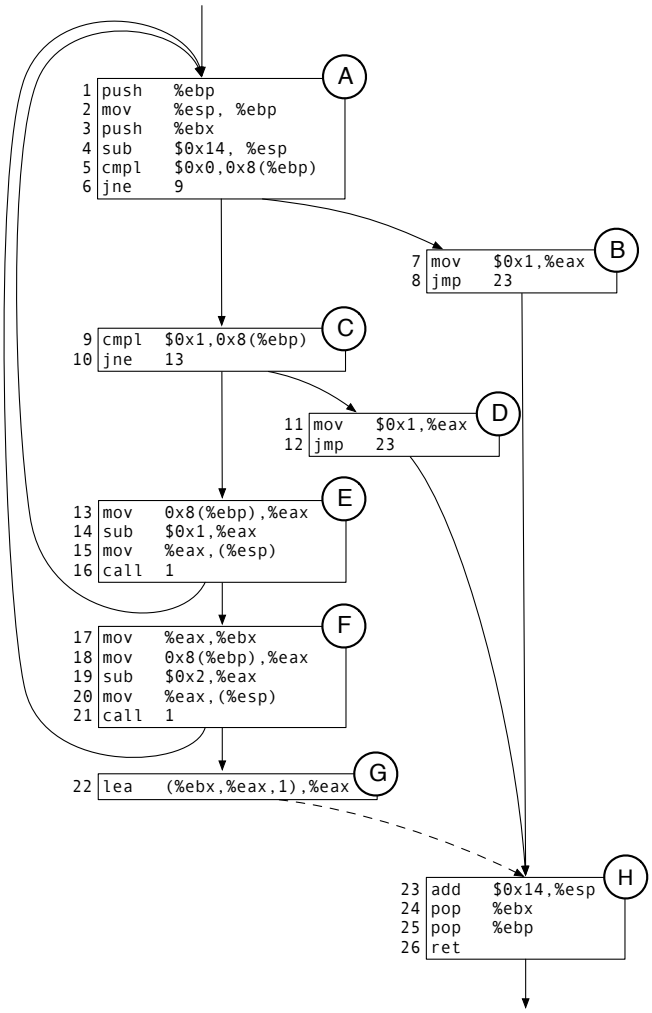


Figure 2.2: To illustrate how basic blocks will be created by tracking branch instructions, a non-optimized version of Fibonacci, Listings 2.1, has been decompiled and divided into basic blocks. Solid arrows points where the instruction pointer, IP, will point after a branch instruction. If an interrupt using precise events based sampling is triggered at D_{12}, the saved IP will point to H_{23}. Dashed arrows show where the control flow falls through.

vector. They showed that a vector with 32 counters is sufficient large to distinguish between most phases.

In this work, we use the first instruction in the basic block as the address. For example, with a simple modulo operator as a hash function, if the size of the BBV is 32 and basic block H is going to be executed, 24 modulo 32 is calculated and used to index into the vector, and the 24th counter is incremented. We will refer to the basic block vector as a signature.

Figure 2.1 show a visual representation of the signature when *gcc/166* is executed. The y-axis shows the counters in the signature, and the x-axis show the time in number of instructions. The intensity shows the value of the counters. White means that the basic blocks for that counter were never executed, and the darker the intensity is the more times the basic blocks were executed. For example, phase *B* spent most of its time in the basic blocks that correspond to the first counter in the signature. We also see that phase *B* has a very stable behavior in all the performance metrics and that changes in the code corresponds well to changes in the other metrics.

Tracking phases by code produces very distinct phases. Consider phase *B* and *C*, if we were to classify phases by comparing the CPI we might consider phase *B* and *C* as belonging to the same phase. If we now look at level 1 data cache hits we see that there is a big difference and phase *B* and *C* should not belong to the same phase. Another interesting pattern is the duration of the phases. Consider phase *C*, it occurs two times, both with very similar duration. If we used CPI again we would have trouble at predicting how long the durations will be, as we cannot distinguish phase *B* from *C*.

The size of the basic blocks can differ, for example, block *A* in the control flow graphs has six instructions while block *B* has only two. We want to measure what code is being executed and where program is spending time. A problem here is that we only increment the counters in the basic block signatures by one when a basic block is executed. If we have two counters that are equal we expect that the program spent equal amount of time in the two basic blocks. This is only true if the blocks have executed the same number of CPU cycles. Lau et al. [18] showed a marginal improvement when the counters were incremented with the size of the basic block.

3 Method

3.1 Phase Capture

To capture program phases we track which basic blocks are being executed. Various methods have been proposed to collect basic blocks signatures, both online [32] and offline [2, 19, 5]. For example, the program binary can be dynamically instrumented using tools such as Pin [4]. Pin allows the user to insert extra code at each basic block that count the number of times it has been executed. However, the runtime overhead of using dynamic instrumentation can be significant [34]. In order to reduce this overhead, Sherwood et al. [32] proposed a hardware extension that can track basic blocks in real time with virtually no overhead but reported no overhead numbers. Others [2, 19, 5] have investigated using VTune [1] and hardware performance counters to sample instruction execution counts in order to collect what they call Extended Instruction Pointer Vector. This approach can be used on existing hardware. One major difference with our work, is that we sample basic blocks execution counts and collect basic block signatures. Lau et al. [18, 19] have shown that tracking the code by larger control flow structures (such as basic blocks) produce better results.

3.1.1 Performance Counters

Modern processors include special-purpose registers for collecting run-time statistics. For example, a counter can be set up to count the number of retired branch instructions. The counter can either be read or programmed to generate a interrupt when the counter overflows.

We used a Nehalem [15, 21] based system that has one performance monitor unit (PMU) per core and four counters per active thread. Operating system support is needed to program the performance counters. Linux *perf_events* is a kernel interface for monitoring different performance metrics, both software and hardware related events. It has been available in the mainline kernel since 2.6.33. When an interrupt occurs due to a counter overflow a SIGIO signal is sent to the program.

On a counter overflow Linux perf_events can record the *instruction pointer* (IP). However the processor will not stop immediately. There is a time delay called *skid* between when the counter overflows and the delivery of the interrupt during which the processor will continue to execute instructions. The recorded IP will thus not point to the instruction that caused the overflow. As instructions take different number of cycles to complete the IP will be biased and point more

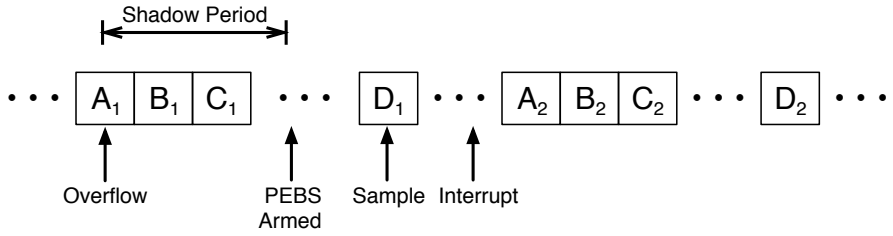


Figure 3.1: There is a time delay between counter overflow and the arming of PEBS hardware. This can lead to bias. The diagram shows a PEBS event. The counter overflows at A_1. The shadow period is the time delay between counter overflow and when PEBS become ready. The next event is then sampled, D_1 and a interrupt is generated.

often to the instruction after a long latency instruction.

As a solution to this problem Intel implements *Precise Event Based Sampling* (PEBS). When the counter overflows PEBS can record the state of the registers, in particular the IP. The IP points to the next instruction, meaning that the IP will point to the start of a basic block. Linux perf_events can also use *Branch Trace Store* (BTS), to compensate and record the IP of the actual instruction that caused the overflow.

We revisit Fibonacci's control flow in Figure 2.2. Solid arrows starts at a branch instruction and end at a basic block. For example, if an overflow happens at A_6 and PEBS is used, the IP will point to either B_7 or C_9 depending on which branch was taken. If BTS is used the IP will point to A_6 and we can sample the end of a basic block.

PEBS solves the skid problem, however depending on the situation we might see some unexpected results. PEBS mechanism goes through four steps when recording the IP. For example, when counting branch instructions:

1. A branch is executed and the performance counter overflows.
2. The PEBS hardware is activated.
3. The next branch triggers the PEBS hardware and the state of the registers are written to the debug store.
4. Finally an interrupt is generated.

There is a time delay between step 1 and 2. Levinthal [21] calls this shadowing. If the duration of a sequence of branches is shorter then the shadowing period, a bias will be introduced that will distort the measurement. For example, consider the sequence of branches in Figure 3.1, there are three branches, A, B, C that occur very rapidly and a branch D that happen after longer period of time. If the counter overflowed at A, B, or C, the PEBS hardware will be activated after the branches have been executed and branch D will be sampled. When the counter overflows at D the PEBS hardware will have time to activate and branch A will be sampled. We want all the four events in the sequence to be uniformly sampled. What really happens is that only branch A and D will be sampled, 25% of the samples will hit branch A and 75% will hit branch D.

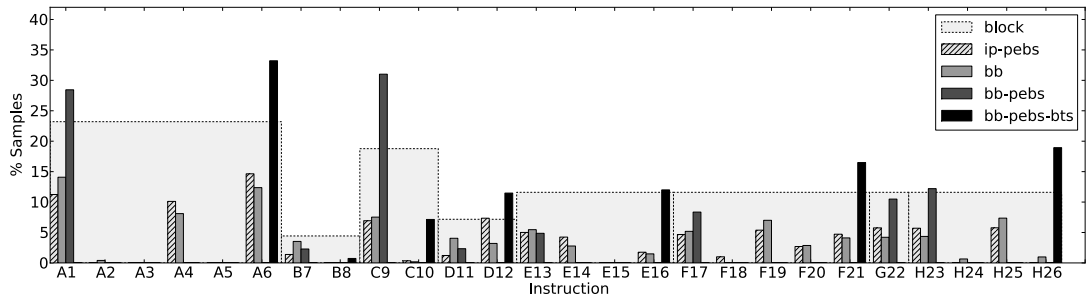


Figure 3.2: Different sample strategies will create different signatures. We have plotted how often each instruction in figure 2.2 are sampled when calculating Fibonacci(42). The x-axis shows the instructions and the y-axis how often they were sampled in percent. Instruction sampling with PEBS(ip-pebs), branch instruction sampling with no PEBS support(bb), with PEBS support(bb-pebs) and with PEBS and BTS support(bb-pebs-bts). The dotted lines how often each basic block should be sampled.

As of this writing, Linux perf_events only support PEBS a small set of performance counters. We had to add one line of code to the Linux kernel to enable PEBS support for BR_INST_RETIRED.

3.1.2 Forming a Signature

To capture signatures we first divide the execution into non-overlapping uniform execution intervals. A performance counter is set to overflow at regular intervals. During each interval another counter is used to sample basic blocks. The counter is incremented each time a branch instruction is executed. The *sample period* is the number of branch instructions between counter overflows. For example, with a sample period of 10 thousand, every 10 thousand branch instruction will generate an PEBS event, the state of the register are written to the *debug store* and the kernel collect the IP and store it in a memory-mapped file.

At the end of every interval, the recorded IPs in the memory-mapped file are used to hash into the signature and increment the corresponding counters. We use *random projection* [32] to reduce the dimensionality of the 64 bit instruction pointer to 5 bits which can be directly mapped into the 32 entries large signature. Random projection, equation 3.1, work by multiplying the IP with a matrix with random values where the sum of each column is equal to one.

$$IP_{64} * M_{64,5} = Index_5 \tag{3.1}$$

It is important to note that we sample branch instructions and capture basic block signatures. [2, 5] also uses performance counters to create signatures, but they sample instruction pointers. A basic block contains multiple instructions. If we only sample instruction pointers we are going to hit the same basic block multiple times but increment different counters. This will just add redundant data and we need to sample more often to achieve the same accuracy compared to basic blocks. To address this problem, Perelman et al. [28] created what they call a *Sampled Basic Block Vector*. They first used Pin [4] to map IPs to basic blocks in the program binary.

They then used the basic block mapping to convert the instruction pointer vector to a basic block vector.

We have plotted different sample strategies for the Fibonacci function in Listing 2.2 and Figure 3.2. The x-axis shows the IPs and the y-axis how often they were sampled in percent. We tested instruction sampling with PEBS (ip-pebs), branch instruction sampling with no PEBS support (bb), with PEBS support (bb-pebs) and with PEBS and BTS support (bb-pebs-bts). The dotted lines and gray area shows the basic block and how often the block should theoretically be sampled. We see that sampling instruction pointers and branch instructions without PEBS produce similar results. With PEBS each basic block is only sampled once but a bias is introduced due to shadowing of the smaller basic blocks.

Sampling basic blocks with PEBS does not produce an exact image of how the code is being executed. Shadowing is relatively deterministic and we can still use the basic block vector as a signature even if it does not depict the true basic block distribution.

3.1.3 Randomization

The sample method we have described so far uses systematic sampling that can be vulnerable to periodic behavior in the execution. To avoid this, we can sample branch instructions at random sample periods. An exponential distribution describes the time between events where events occur continuously and independent of each other at a constant average rate. After a basic block has been sampled, the sample period is updated by randomly selecting a value from an exponential distribution. The rate parameter λ is chosen so that the expected value is equal to the sample period.

$$\mathrm{E}[X] = \frac{1}{\lambda} = \textit{sample period} \;\Rightarrow\; \lambda = \frac{1}{\textit{period}} \tag{3.2}$$

We let the kernel buffer all the basic block samples during a execution interval in order to reduce the number of context switches. Thus, we can not update the sample period from user space and we had to add randomization support to perf_events. The Linux kernel optimizes context switches by not saving and restoring floating point registers at context switches. To handle this an array of fixed point pre-calculated exponential distributed variates were generated. A new sample period is generated at each interrupt by randomly selecting a variate from the array and scaling it to the correct size.

3.1.4 Dynamic Intervals

Most phases span over several intervals. In Figure 2.1, phase *C* spans over 50 intervals. In these cases we could boost the performance by increasing the size of the interval while decreasing the sample rate.

If two consecutive intervals are in the same phase we dynamically increase the size of the interval by a user defined scaling factor. The size of the interval is reset to the base size when a phase change occurs. We also take advantage of the repeatable behavior of programs. If we

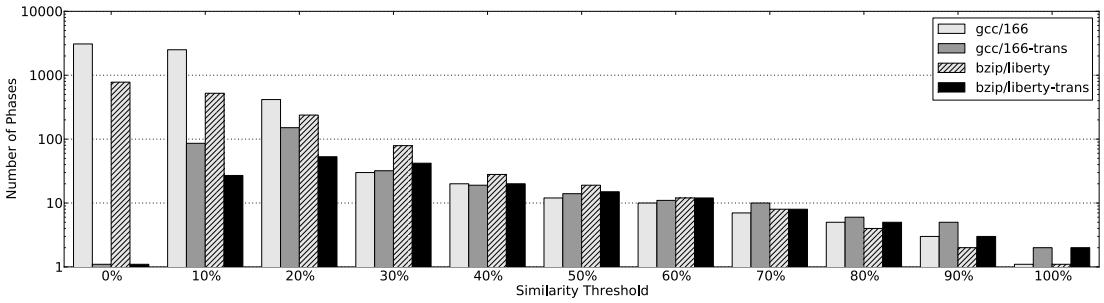


Figure 3.3: The similarity threshold can affect the number of phase we detect. We have plotted the execution of *bzip/liberty* and *gcc/166* with different thresholds and measured how many phases we detect. *bzip/liberty-trans* and *gcc/166-trans* show the number of phases when the transition threshold is set to two.

enter a phase that we know spans over several intervals we increase the interval directly to the size used the last time we encountered the phase.

This boosts our performance, but it also lowers the accuracy, each time we increase the interval we will overshoot when the phase is over. For example, if the interval has been increase up to ten times we might miss ten phases.

3.2 Classification

Two intervals of execution are rarely going to have the exact same signature. We need a way to classify similar signatures into the same phase. In this section we evaluate two ways to accomplish this, distance based classification and a sequential version of K-means clustering algorithm.

3.2.1 Distance Based Classification

Two signatures are classified into the same phase if the distance between them are below a given *similarity threshold*. First the signatures are normalized. If the signatures are not normalized the magnitude would be effected by the sample period and size of the interval.

$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i| \tag{3.3}$$

We use the Manhattan distance, equation 3.3, to calculate the distance between two signatures. The maximum distance between two normalized vectors are two. We can define a simple threshold in percent of the maximum distance. If the distance between two signatures are smaller then the similarity threshold they are classified as belonging to the same phase.

For online phase classification, we use a *least recently used* (LRU) cache to hold the most recently encountered phases. Sherwood et al. [32] showed that a 32 entries large cache is sufficient for most programs. Each entry contains a phase id and a signature. When an interval is over, the signature is compared with the signatures in the cache. The closest entry is selected. If the distance is below the threshold, we replace the old signature with the new and classify the

interval with the phase id in the entry. If the distance is above the threshold for all cached entries, we invalidate the oldest entry and generate a new phase id.

It is important to select a good threshold. To show what happens when the threshold is varied we have plotted the execution of *gcc/166* and *bzip/liberty* in Figure 3.3. We varied the threshold and measured how many phases we detected. We can see that for small thresholds we get a lot of phases. In the extreme, if the threshold is set to zero every interval will be counted as a unique phase. The opposite is also true; if the threshold is set to 100% the whole execution will be counted as one big phase.

When a program transition from one phase into another within an execution interval, the signature will contain information from two different phases. These *transition phases* rarely happen but when they do they pollute the cache. To avoid this Lau et al. [20] added a counter to each cache entry. When a new signature is inserted it gets a common transition phase id. Each time a signature is classified into an existing entry the counter is incremented. Only when the counter goes above the *transition threshold* is a new phase id assigned to the entry.

To show how this can reduce the number of phases we plotted the execution of *gcc/166* and *bzip/liberty* in Figure 3.3. The transition threshold is set to two, i.e. each unique phase must have occurred at least two times. When the similarity threshold is set to 10% and we use a transition threshold of two the number of phases is significantly reduced.

The purpose of the classifier is to classify similar intervals of execution into the same phase. We want each phase to exhibit a homogeneous behavior across all the intervals it occurs in. Depending on how we choose the threshold we will change how homogeneous the phases are. Decreasing the similarity threshold will increase the number of phases but the homogeneity in each phase is better. This must be considered when selecting a similarity threshold.

3.2.2 Sequential K-Means

We have seen that it can at times be difficult to pick a similarity threshold, and different programs need different thresholds. Sometimes it might be desirable to just pick a upper bound on how many phases we can have. K-means [23] is a common algorithm in machine learning to find clusters in data. It is an offline iterative process that can be divided into four steps:

1. K means are randomly selected from the data set.
2. K clusters are created by assigning each signature to the nearest mean.
3. K new means are then created by calculating the mean from the signatures in each cluster.
4. Step 2 and 3 are repeated until a convergence criteria is met.
5. Assign each cluster a unique phase id.

Using K-Means introduces two problems. It is iterative and requires that we know all the signatures in advance. We can change the above steps and make an online version [8]. Step 1, a distance based classifier is used until we have K unique phases. Each phase is used as a starting position for the means. Step 2 and 3, the nearest mean is selected when a new signature is being

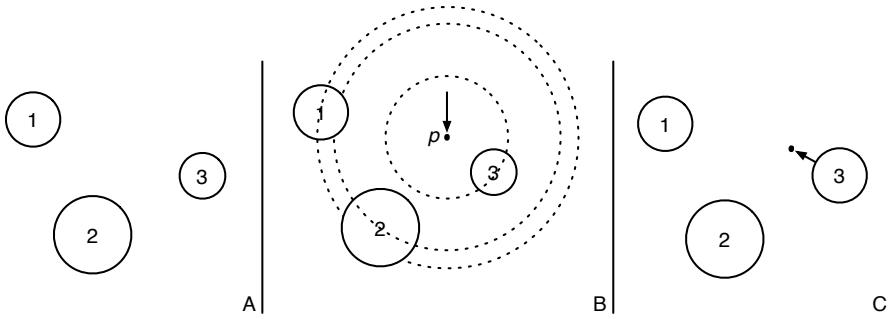


Figure 3.4: Sequential K-means can be used for arbitrary many dimension. We have plotted the algorithm in a 2D space, but the same algorithm can be applied to higher dimensions. Figure A shows three means, m_1, m_2 and m_3. The radius of the means describes the weight. In B, a new point p is inserted. The distance from p to each mean is calculated and the closest mean is chosen, m_3. In C, m_3 is updated by moving is closer to p and incrementing the weight.

classified. The mean is moved closer to the signature with a learning factor, Equation 3.4. Each time a signature is assigned to the mean the weight of the mean is increased. This process is repeated for every signature. An important property with sequential K-Means compared to the standard K-Means is that the signature can be discarded afterwards, this results in a constant memory usage. For the standard algorithm we have to remember all the signatures when the new means are calculated in step 3.

Figure 3.4 show how sequential K-means works in a 2D space. Figure A shows three means, m_1, m_2 and m_3. The radius of the means corresponds to the weight. In B, a new point p is inserted. The distance from p to each mean is calculated and the closest mean is chosen, m_3. In C, m_3 is updated by moving is closer to p and incrementing the weight.

$$m_3 = m_3 + \frac{p - m_3}{weight_3} \tag{3.4}$$

One weakness with sequential K-means is that we need a good estimate of how many phases a program has. We also have a learning period when the means are moved into place. All the means will be clustered around single point during the learning period. We will then have intervals belonging to the wrong phase when the means start to move apart.

3.3 Prediction

Programs exhibit repetitive phase behavior. The same sequence of phases repeats to form larger macro phases. We can take advantage of this to predict which phase the program will enter next. This can be used to start different optimization or prepare resources a head of time instead of just reacting when a the program behavior change. In this section we discuss several methods to predict what phase the next interval will belong to.

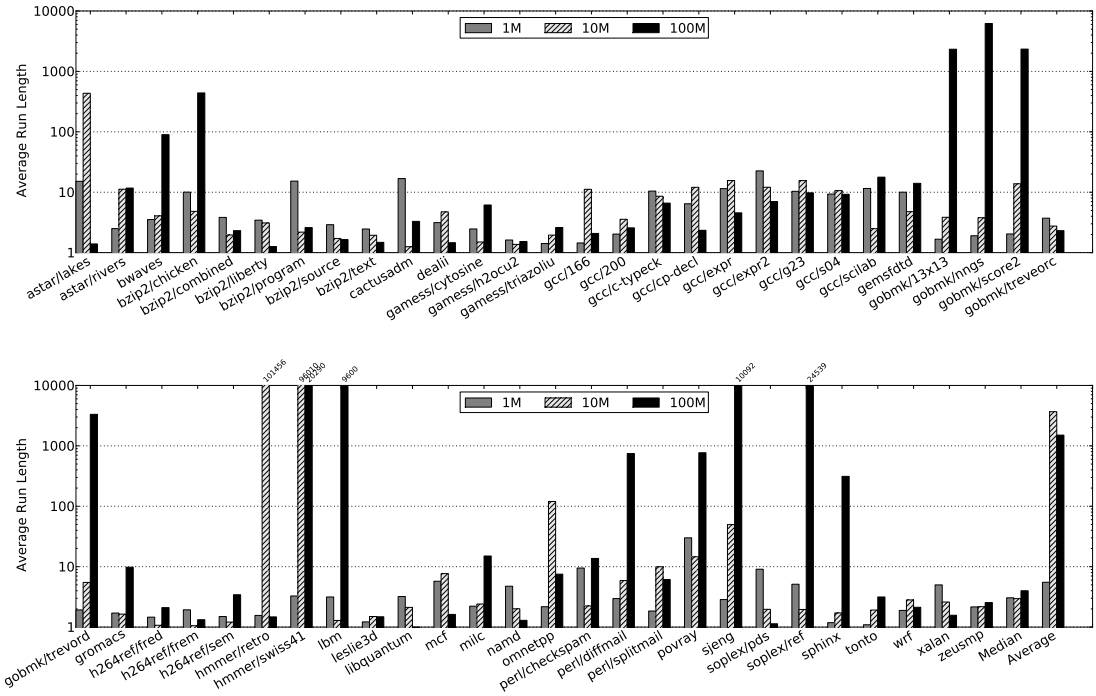


Figure 3.5: The size of the interval can affect the phase classification. We ran the SPEC applications with different interval sizes. The average run length is the number of consecutive intervals belonging to the same phase. A short run length shows that we have a lot of phase changes.

3.3.1 Last Value Prediction

The last value predictor [32] is the most basic predictor. It will always predict that the next interval will belong to the same phase as the previous interval. Figure 3.5 show the average run length for the SPEC applications. The run length is the number of consecutive intervals belonging to the same phase. We see that most phases have a duration that spans over several intervals leading to a good prediction rate for stable periods. The last value predictor will only misspredict at a phase change.

3.3.2 Markov Prediction

A Markov-N predictor [32] uses an N interval long phase history to index into a cache. Figure 3.6 shows a Markov predictor where N is equal to two. The cache contains a phase history and the phase id of the interval that followed the history the last time it was observed. First, a learning step is done where the predictor stores a previously seen phase history. In Figure 3.6, phase history $A[5,2]$ is used to index into the cache and the phase id of the interval after the history is saved as a prediction, i.e. 3. Second, in the prediction step, the last execution intervals are used to index into the cache and look up the prediction. In Figure 3.6, phase history B is used

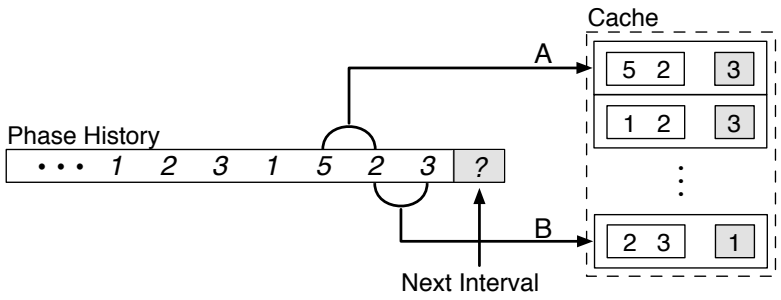


Figure 3.6: The Markov predictor uses phase history to index into a cache. First a learning step where the predictor stores a previously seen history *A* with the phase after the history. In the prediction step it uses history *B* to index into the cache. If the cache entry is valid we predict the same phase as was seen last time after the history was observed otherwise we fall back to last value prediction.

and we predict that the next interval will belong to phase 1. If the cache entry is invalid we fall back to last value prediction.

The second predictor we tested was a Markov Partial Pattern Match Predictor. It consists of the set of Markov-N predictors. When a prediction is made we first do a prediction using a phase history of N intervals, each time a prediction fail we use a smaller history. If no phase history matched we fall back on a last value prediction.

Finally we tested a Run Length Predictor [32]. A run length is a tuple that consists of a phase id and how many consecutive intervals the phase had when it was last observed. The run length is then hashed and used to index into the cache the same way as a Markov predictor. One advantage of the Run Length predictor is that it can be used for phases that span over longer periods of time. The Markov predictor is bound to the size of the history. For example a Markov-2 predictor can only see phase history that span two intervals while a Run Length predictor can see arbitrary long phases. On the other hand a partial pattern match predictor can find more complex patterns.

It is important to consider all aspects when we choose a predictor. In some cases a Markov predictor might have a overall better prediction rate, but it might predict false phase changes. On the other hand using a last value predictor results in a reactive system. Great care should be used when picking a predictor and how it affects performance.

4 Evaluation

4.1 Experimental Setup

Software	
Kernel	Linux 2.6.37 - git
GCC	4.4.3
Hardware	
System	HP Z600 Workstation
Memory	6 GB ECC
Processor	Intel Xeon E5620@2.40GHz
Architecture	x86_64
Threads per core	2
Cores per socket	4
CPU sockets	1
NUMA nodes	1
CPU MHz	2395
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	12288K

Table 4.1: Experimental Setup

The evaluation was done on a Nehalem based system, Table 4.1, with the CPU SPEC 2006 [12] benchmark suite. *calculix* was not used due to time and space constraints.

4.2 Methodology

Previous research [32, 18, 20] have focused on evaluating the intra-phase homogeneity. Each phase should have a similar performance metric for all the intervals it occurs in. Cycles per instructions have been used as it reflects changes in different metrics. We will use the same

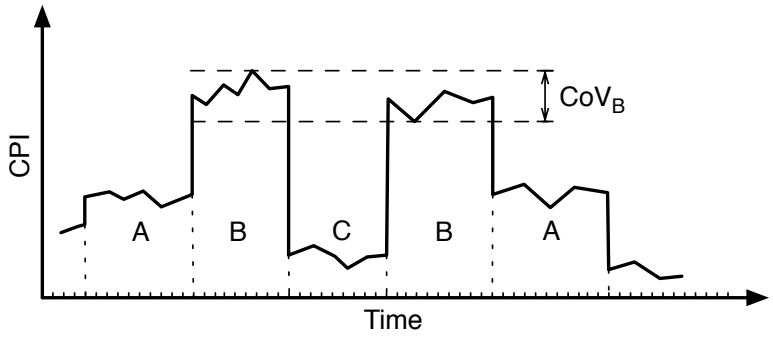


Figure 4.1: Two intervals belonging to the same phase will rarely have the same CPI. The plot show a simple example of how the CPI can vary over time. We have three phases *A*, *B* and *C*. The intra-phase variance changes depending on the number of phases we detect. We want few phases with low intra-phase variance.

approach to evaluate how effective our classification is. To measure the homogeneity we used an extra performance counter to sample the number of CPU cycles executed during each interval.

Figure 4.1 shows how the CPI can change overtime for a fictional program. In terms of CPI, it has three distinct phases, *A*, *B* and *C*. However, there are still variations in the CPI between execution intervals that are classified to belong to the same phase. Since the goal of phase classification is to group execution intervals with similar behavior (in this case CPI), these intra-phase variations are not desirable. To measure the quality of our phase classification we use the Coefficient of Variance (CoV), which is the standard deviation of the CPI across the execution intervals divided the average. If all the execution intervals that belong to the same phase have exactly same CPI the CoV will be zero, while a non-zero CoV indicates that there is some amount of intra-phase variation.

A large CoV and a small number of detected phases may indicate that the similarity threshold is to high, i.e. the intervals in phase *A* and *B* has been classified into the same phase resulting in a large intra-phase variation. A low CoV and a large number of phases shows that the similarity threshold is too low, i.e. the interval *B* has been split into multiple smaller phases. We want a similarity threshold that produces few phases with a low CoV.

4.3 Phase Capture and Classification

4.3.1 Picking a threshold

The similarity threshold determine how much two intervals can deviate from each other without being classified into separate phases. Changes in the threshold will affect the intra-phase variance. When the threshold is set to zero, every interval is classified as a unique phase and we have no intra phase variance. When the threshold is 100% the whole program is considered as a single phase and the Figure shows the CPI variance for the whole program.

Figure 4.2 show the average CoV for all the applications with different similarity thresh-

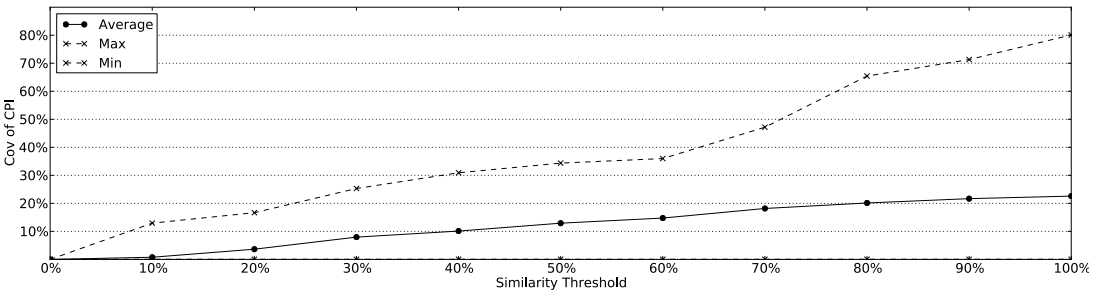


Figure 4.2: The similarity threshold affects the intra-phase homogeneity. We plotted the CoV for the SPEC applications. All the application were run with a 100M instructions interval with a 100K sample period. The solid line shows the average CoV and dashed the minimum and maximum CoV. The minimum CoV is hidden by the x-axis.

olds. We used 100 million instructions intervals and a sample period of 100 thousand branch instructions. Solid lines show the average CoV and dashed the minimum and maximum CoV. The minimum CoV is at the bottom. The difference between the maximum and minimum CoV is quite large, the CoV can vary a lot from application to application. For example, *gcc/166* has a Cov of 71.3% for the whole program while *hmmer/retro* has a low CoV at 0.04%. Picking a threshold depending on the target application can significantly improve the CoV.

4.3.2 Phase Granularity

The minimum length of a phase is restricted to the size of the interval, a short interval results in smaller phases. If the interval is too long, everything will appear as transition phases as each signature will contain samples from multiple smaller phases.

We set the similarity threshold to 50% and measured the average run length, i.e. the number of consecutive intervals belonging to the same phase. We tested interval sizes of 1M, 10M and 100M instructions. We used a sample period of 1K, 10K, 100K to keep the number of samples per intervals at a constant rate.

Figure 3.5 show the average run length for the benchmarks and Figure 4.3 show the number of phases we detect. A closer look at *gcc/166* show that the number of detected phases differ a lot depending on the size of the interval. The number of detected phases for 1M, 10M and 100 million was 750, 57, 19 respectively. The average run length was 1.44, 11,17 and 2.07.

Intuitively one might expect that the run length would be larger for shorter intervals. If we have a run length consisting of two intervals at 100 million instructions granularity we would think that we should find a run length of 20 intervals at 10 million instructions granularity. This is not the case; we can see that the number of phases increases in Figure 4.3 with shorter intervals leading to more phase changes and shorter run lengths.

To sample every one thousand branch instruction needed for a interval size of one million instructions incurs a significant overhead compared to 100 million. The average overhead for 1M, 10M and 100 million is 76%, 8.4% and 1.2% respectively.

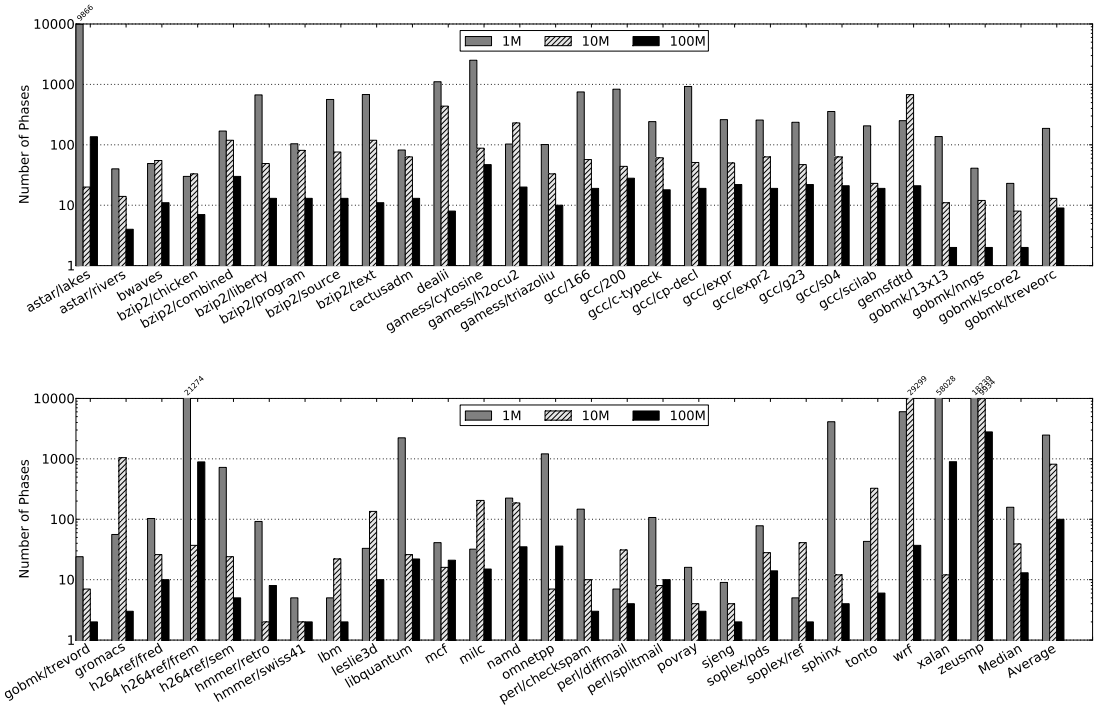


Figure 4.3: The size of the interval affects the number of phases we detect. We ran the SPEC applications with different interval sizes and recorded how many phases we detected. We can see that we find more micro-phases when we decrease the interval size.

4.3.3 Dynamic Intervals

We have observed that several programs have long run lengths. If we enter a long phase we can increase the interval size and sample period in order to lower the overhead. However, when the interval become too large we might classify intervals into the same phase that we would otherwise not do. This can lead to false phases that would not occur with a fixed interval size.

It is important to understand the benefits and how much accuracy we sacrifices. We ran the SPEC applications with dynamic intervals enabled. The base interval size was set to 100 million instructions and the sample period was 100 thousand. We set the upper bound to 15 meaning the interval cannot grow larger than 15 intervals. We then divided the intervals into the base lengths and used the CPI data from the tests with fixed intervals.

We plotted the result in Figure 4.10. The black bars show the CoV when a fixed interval size is used and the dashed when dynamic intervals are used. The gray bars show the overhead for fixed intervals and the bars with circles in them show the overhead for dynamic intervals.

We see some mixed results. For *bzip2/chicken* the CoV is nearly doubled when dynamic interval is used. The reason for this is that *bzip2/chicken* has two very distinct phases with very different CPI. We call the two phases A and B. They appear in a specific pattern, for example, $A_1 - A_2 - A_3 - B_4 - A_5 - A_6 - A_7 - B_8$. We increase the interval after A_1 and A_2. The two intervals

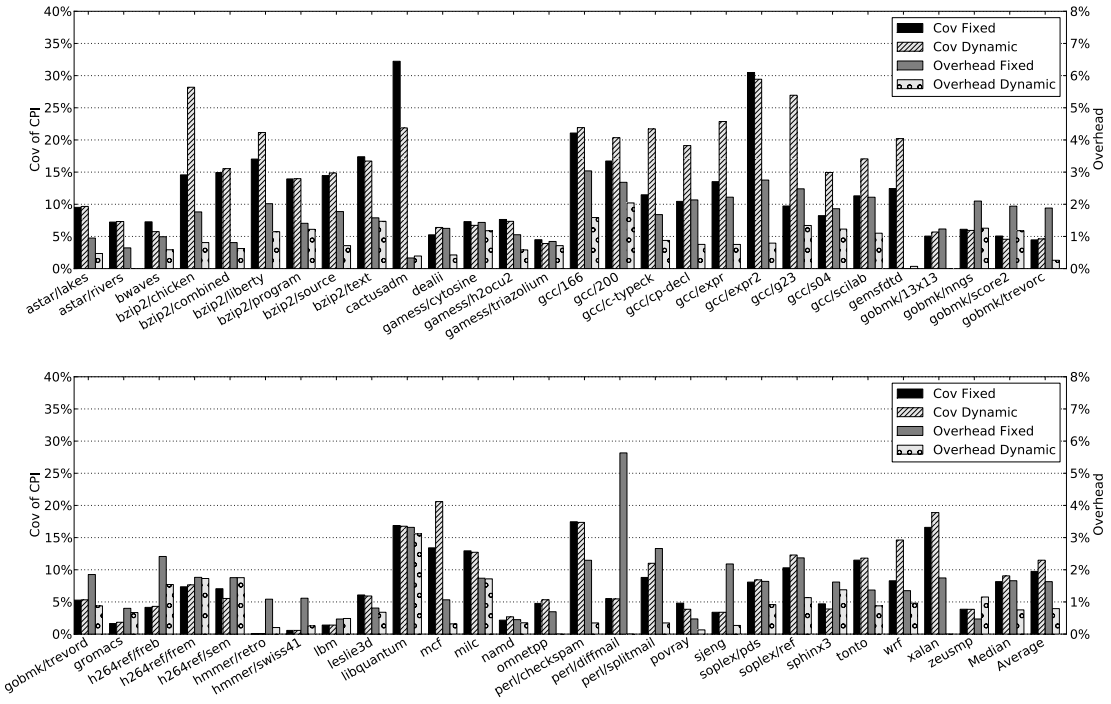


Figure 4.4: The CoV for fixed intervals and dynamic intervals, and the overhead.

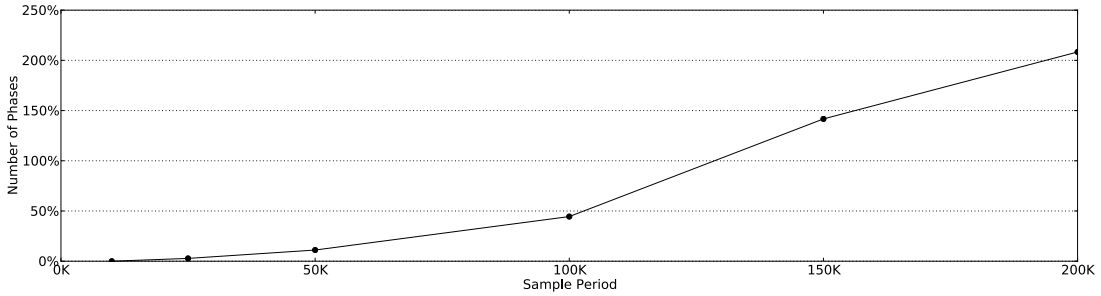
A_3 and B_4 will then be merged and classified as a new phase. This also results in a phase change and the interval is reset. The phase that contains phase A_3 and B_4 will have a very high CoV.

We see a tiny increase in CoV for *gcc/166* but a significant improvement in overhead. If we look at Figure 2.1 we see that *gcc/166* has a set of very long phases and few phase changes. The biggest problem with dynamic intervals is phase changes and we see a much better result with programs with few phase changes. On average the overhead is 49% lower with a 18% increase in CoV.

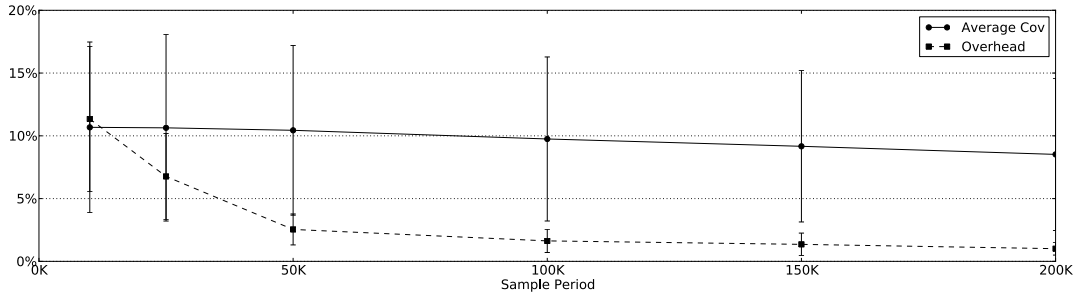
4.3.4 Accuracy vs Overhead

We have seen that we can detect phases by sampling basic blocks. The accuracy depends on how often we sample. The highest accuracy would be if we could sample every basic block, this, however would result in a large execution time overhead when collecting signatures. We need to understand the relationship between sample period and accuracy in order to choose the right sample period.

We ran the SPEC applications with different sample periods. The similarity threshold was set to 35% and the transition threshold was set to two intervals, meaning that a new phase id is only created if a signature has been seen more than two times. We tested sample periods of 10, 50, 100, 150 and 200 thousand. We measured the overhead, average CoV, and the median number of phases we detect. The CoV was calculated with data obtained from the test runs with



(a) The figure shows the median number of unique phases that was detected at the different sample periods. We used the number of phases that was detected at 10K sample period as the baseline. The y-axis show how many more phases that was detected compared to the baseline.



(b) The figure shows the how CoV is affected when the sample period changes. The solid line shows the average CoV for the SPEC applications and the dashed line shows the average overhead. The vertical bars show the standard deviation.

Figure 4.5: The sample period affects the phase classification. We ran the SPEC application with different sample periods and recorded how many phases we detected. The number of phases increases with the sample period. As a result we have a marginal improvement in CoV.

that used a 100K sample period.

Figure 4.5(a) shows the median number of unique phases that was detected at the different sample periods. We used the number of phases that was detected at 10K sample period as the baseline. The y-axis show how many more phases that was detect compared to the baseline. We see that the number of detected phases increases with the sample period, i.e., more false phases are detected. The program does not contain any more phases. These additional phases are a result of sampling and loss of information.

Figure 4.5(b) shows the average CoV at different sample periods. The solid line shows the average CoV and the dashed line show the average overhead. The vertical bars show the standard deviation. The CoV is lowered as a result of the additional phases. It is very important to keep the number of phases low if we want to use SimPoint [31] to minimize simulation time and simulate each phase once. On the other hand if the amount of work per phase is relatively low it might be acceptable with a couple of redundant phases.

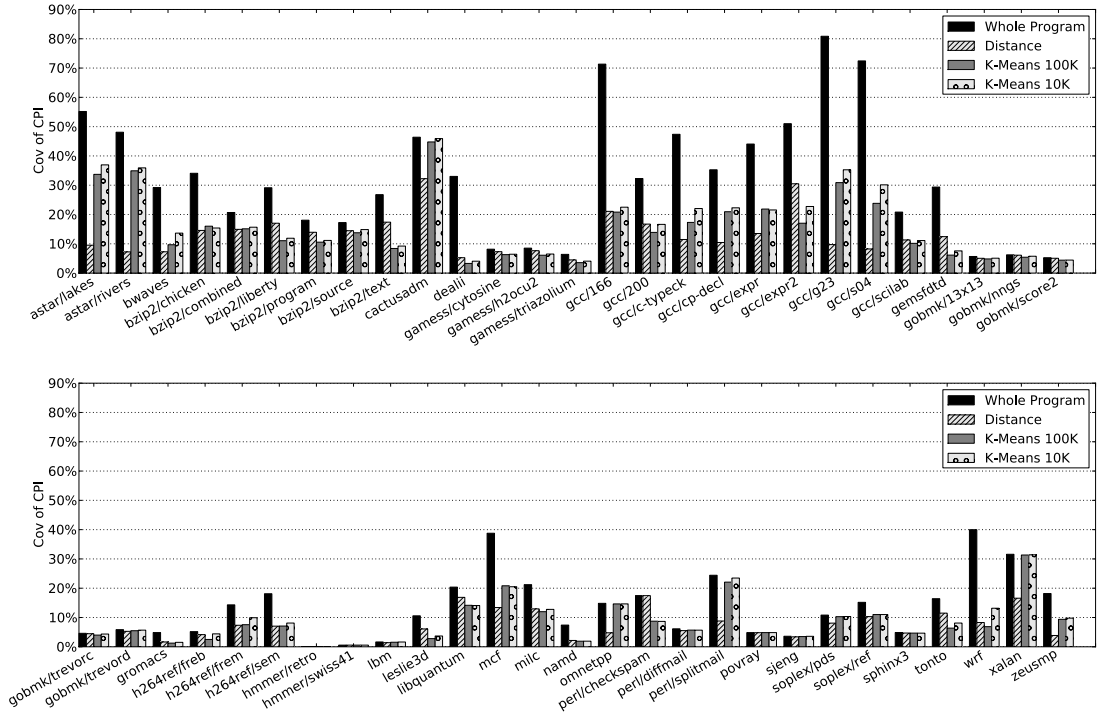


Figure 4.6: It is important to choose a good classifier. The graph chows the average CoV for the SPEC application with different classifiers. The black bars show the CoV when the whole program is considered as a single phase. *K-Means 100K* shows the result of sequential K-Means when the *K* parameter based on the number of detect phases when distance based classification was used with a 100 thousand sample period.

4.3.5 Sequential K-Mean Classification

Choosing the right similarity threshold is crucial to achieve accurate phase classification and prediction. A low threshold result in more phases while a high threshold increases the intra-phase variance. We can use sequential K-means instead of relying on the similarity threshold.

We first used distance based classification with 35% similarity threshold and two interval long transition threshold and measured the number of detected phases. The SPEC applications were then run with K-means classification where *K* was set to the number of detected phases with distance based classification.

Figure 4.6 show the result, the black bars show the CoV of the whole program. The light dashed bars show the CoV with distance based classification and the other two show the result of the K-means classification. The different K-Means show the result when the *K* value is based on distance based classification with a sample period of 10K and 100K. We saw earlier in from Figure 4.5 that 100K will detect more phases, thus K-Means 100K will have more means compared with 10K.

Important to note here is the whole program CoV between the applications. It is more im-

portant to track program phase behavior for programs with large CoV. Consider SimPoint, if we randomly select a couple of intervals from *hmmer/retro* and simulate them the error would be quite small compared to a whole program simulation. If we do the same for *gcc/166* that have many phases with very large intra-phase variance the error would be quite noticeable.

Distance-based classification has an overall better accuracy. The largest problem with K-means is the learning period. For example *astar/rivers* has a stable start period. During this period all the means will be clustered around the same point. When they later start to spread the earlier intervals will be misclassified. Programs such as *gcc/expr2* has a very short start period with a lot of different phases and we see a much better result.

4.4 Prediction

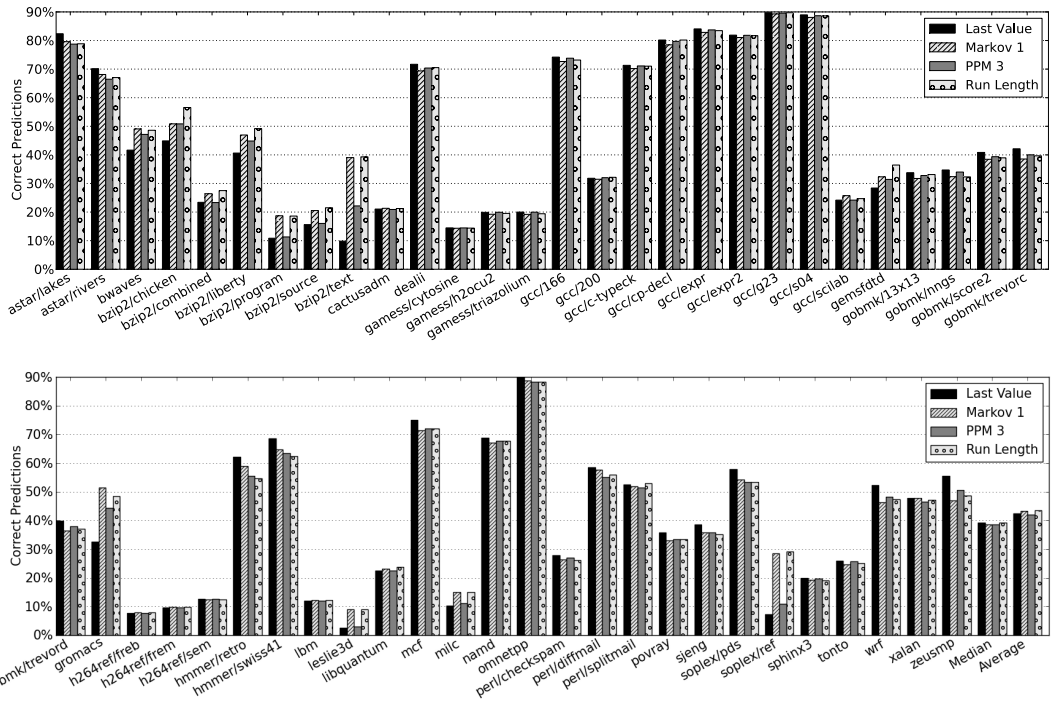


Figure 4.7: The figure shows the number of correct predictions for each predictor. All the SPEC applications were run with a 100M instructions intervals with 100K sample period. Markov 1 shows the result of the Markov predictor with a pattern size of one interval and PPM show the result of Partial Pattern Match with three Markov predictors $\{1,2,3\}$.

Predicting what phase we will enter next can be a powerful tool. For example, we can start to allocate resources in advance so they can be ready when they are needed. It is important that we have a high accuracy as the cost of various optimizations can be high. In this section we evaluate the different predictors we use.

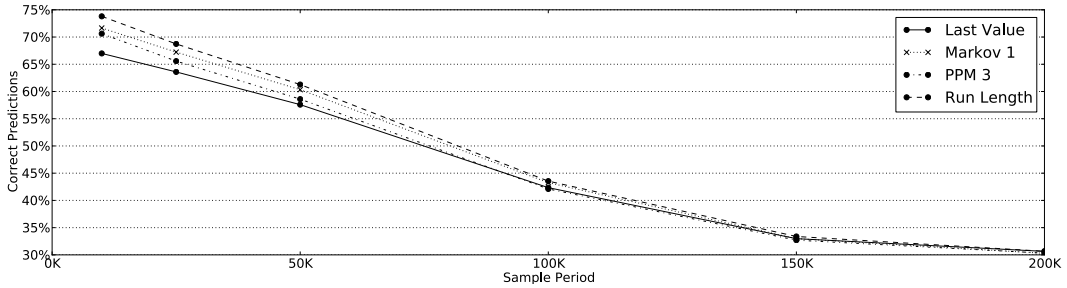


Figure 4.8: The number of detected phases follows the sample period. The false phases degrade the prediction accuracy. The figure shows the average percent of correct predictions at different sample periods. We can see that we loose accuracy very fast.

4.4.1 Prediction Accuracy

Figure 4.7 shows the number of correct predictions, first, the last value predictor, second, a Markov predictor with a pattern size of one interval and third, a Partial Pattern Match predictor with three Markov predictors with pattern size of 1, 2 and 3 intervals. Finally, the run length predictor. All the SPEC applications were run with 100M instructions intervals and a 100K sample period.

It is interesting to note here is that we have a correlation with the intra-phase variance. If we look at the CoV for the whole program in Figure 4.6 and compare it with the prediction accuracy we see that a high intra-phase variation results in good prediction. One reason is that programs with high CoV have very clear phases. For example *gcc/166* in Figure 2.1 has very visible phases and we see very good prediction accuracies. They are also relatively insensitive to the similarity threshold as the signatures differ a lot between phases.

The run length predictor achieved a minor improvement over the last value predictor. Overall the average accuracy is below 50%. We saw earlier that the number of phases increase with the sample period. With a sample period of 100 thousand we have around 50% more phases. These phases are a result of sampling and introduce randomness in the phase pattern. Figure 4.8 shows the average correct predictions for the SPEC applications at different sample periods. The figure shows that we loose accuracy very fast. With a sample period of 10 thousand we have nearly 75% correct predictions, but at 100 thousand the prediction accuracy is below 50%. If we need very reliable predictions we must sacrifice some overhead.

We then examined how the similarity threshold influence predictions and we measured the average predictions for the SPEC applications with different similarity threshold. Figure 4.9 shows the result at similarity thresholds. We use a transition threshold of two intervals, meaning that a phase must have more than two intervals before being assigned a unique phase id. For low similarity thresholds the number of transition phases increases. The average number of transition phases for 0%, 10%, 20%, 30% similarity thresholds are 100%, 78%, 30% and 6% respectively. Due to the number of transition phases it is easy to simply predict that the next interval will be a transition phase. When the number of unique phases increases the prediction accuracy decreases. Choosing a higher similarity threshold leads to better prediction but we increase the intra-phase

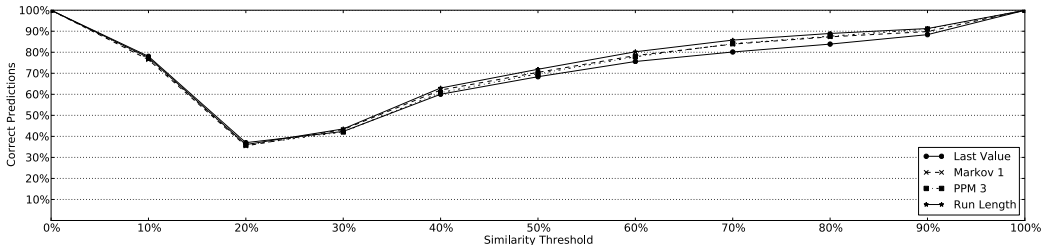


Figure 4.9: To show how the different predictors are affected by the similarity threshold we plotted the average percent of correct predictions with different similarity thresholds. All the SPEC applications were run with a 100M instructions intervals.

homogeneity.

4.4.2 False Phase Change Predictions

Predicting a phase change that never happens can start an expensive optimization. For example, if we use prediction to assign threads to different cores we want to be very sure that we only migrate them when they enter a new phase.

Figure 4.10 shows the percent of predictions that predicted a phase change that never happened. The last value predictor is at the bottom. The program *hmmer* has frequent predictions that can potentially start expensive operations, while *leslie3d* is barely noticeable and very safe. The last value predictor will never predict a false phase change, but it has lower prediction accuracy. The Markov 1 predictor performed worst, and has an average above 5%. The PPM 3 and run length predictor performed better and have an average blow 5%.

4.4.3 Predicting Performance Metrics

The prediction accuracy has so far only considered what phase the next interval will belong to. An incorrect prediction can still be a good representation of the next interval. Consider the additional phases that are detected when the sample period is increased. The extra phases lead to more phase changes, however the changes happen during what would before have been a single a phase. Thus, an incorrect prediction might still predict a phase that have a similar performance metric.

Figure 4.11 shows the predicted CPI error for the run length predictor at different sample periods. In each execution interval we calculated the absolute prediction error, the error between the predicted CPI and the actual CPI for the interval. The predicted CPI is the average CPI (measured so far) for the phase we predicted. The three left bars in Figure 4.11 shows the average error. *bwaves* has a prediction accuracy for 10K, 50K and 100K sample periods of 86.9%, 61.7%, 48.6% respectively, but a relatively constant CPI error. Thus, the incorrect predictions are still predicting phases that accurately represents the next interval.

One of the objectives with program phase detection is that run-time statistics for the whole program are not a good representation for various parts of the execution. We measured the av-

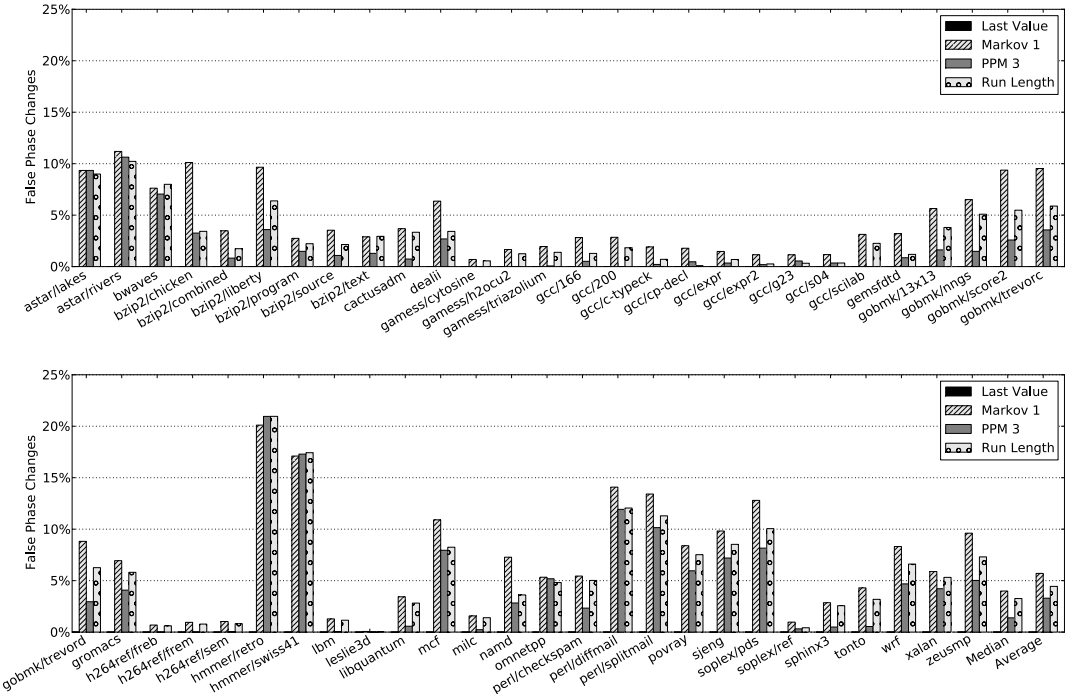


Figure 4.10: Predicting false phase changes can start up costly operation. The figure shows how often the predictors predict a phase change but no phase change occur. All the applications were run with 100M instructions intervals and a 100K sample period.

erage CPI for the whole program and used the average CPI as a prediction. The right bar in Figure 4.11 shows the absolute error when the average CPI is used as a prediction. A large error shows that it is important to consider program phase behavior. For example, *bwaves* has a very high error, meaning that the average CPI is not a good approximation of the program execution. *hmmer/retro* on the other hand has extremely low CPI error and any execution interval can be used to approximate the program behavior for the whole execution.

Comparing the predicted CPI and the average program CPI prediction shows that the prediction is quite accurate. Consider *bwaves*, the difference show that the prediction accuracy is high, regardless of sample period. The average CPI error is below 10% compared with 50% for the average number of correct predictions.

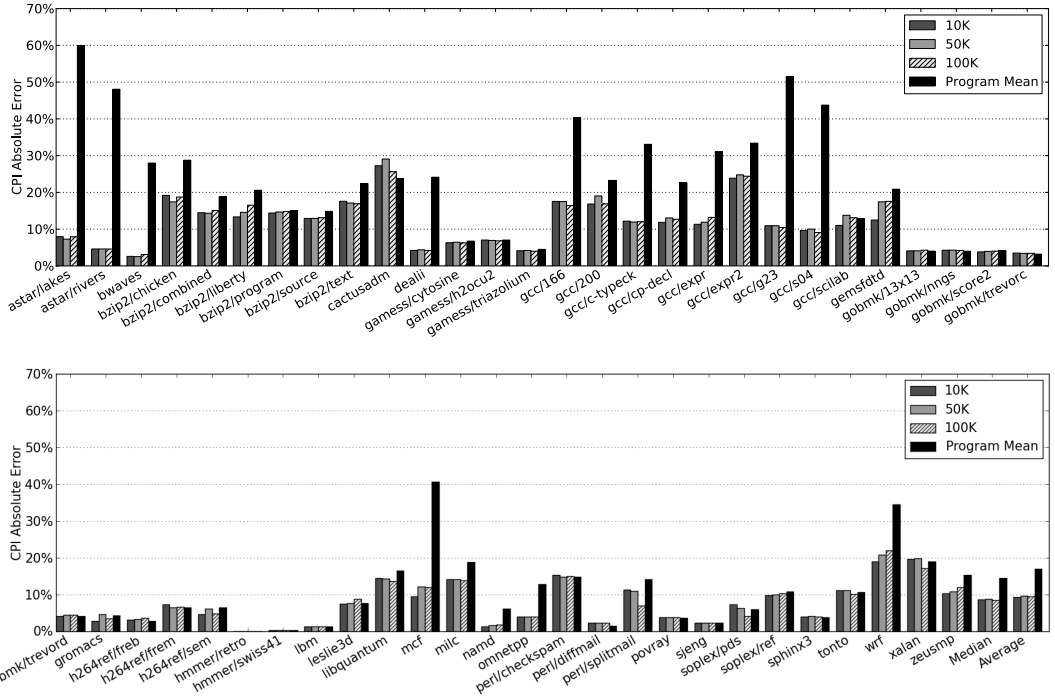


Figure 4.11: A larger sample period leads to a lower prediction accuracy, however an incorrect prediction can still be a good representation. The figure shows the CPI error between the predicted CPI and the actual CPI. Program mean shows the error when the average CPI is used as a prediction for all execution intervals.

5 Conclusion

In this thesis we examine how hardware performance counters can be used to sample basic block vectors and use them to classify and predict program phase behavior. Our evaluation shows that sampling basic blocks can be an effective method to classify phases.

We examined how the phase detection is affected by the size of the interval. We looked at 1, 10 and 100 million instructions, and found more micro phases at shorter interval sizes leading to more phases. As a result the average run-length is shorter. We evaluated the potential performance improvements with dynamically growing intervals. Our evaluation shows that we can reduce the overhead with 49% at an 18% increase in CoV.

The sample period has a huge impact on the phase classification. The number of phases increases with the sample period; as a consequence we see a marginal improvement in intra-phase homogeneity. The extra phases are redundant and only a result of sampling. It is important to take this into account when choosing the right sample period. A high sample rate should be used if the amount of work per phase is high and we need as few phases as possible. On the other hand, we can allow a lower sample rate if the amount of work per phase is low.

The prediction is very susceptible to the sample rate. The prediction accuracy was below 50% for 100 thousand sample period. We saw a steady increase in accuracy when we raised the sample rate. This is due to the additional phases that are created when the sample period increases. We found no real difference in accuracy between the predictors for lower sample rates. At a 10 thousand sample period, the run length had a nearly 75% prediction accuracy while the last value predictor had a little above 65%. The loss in prediction accuracy that is due to sampling does not always result in a bad prediction. An incorrect prediction will usually predict a phase that have a similar performance metric. Changing the similarity threshold improves the accuracy but we increase the intra-phase homogeneity.

Our evaluation shows that we can reliably detect program phases at a very low overhead. We believe that our library can be used to create new and interesting optimizations, and we plan to incorporate our library with others tools.

Bibliography

[1] Intel vtune performance analyzer homepage. URL http://www.intel.com/software/products/vtune/.

[2] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. A. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: http://dx.doi.org/10.1109/MICRO.2004.34.

[3] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W.-m. W. Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 233–244, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1. URL http://portal.acm.org/citation.cfm?id=774861.774887.

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[5] B. Davies, J. Bouguet, M. Polito, and M. Annavaram. ipart : An automated phase analysis and recognition tool. Technical Report IR-TR-2004-1-iPART, Intel Corporation, 2004.

[6] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1605-X.

[7] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 217–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL http://portal.acm.org/citation.cfm?id=956417.956539.

[8] R. O. Duda. Sequential k-means clustering, 2007. URL http://www.cs.princeton.edu/courses/archive/fall08/cos436/Duda/C/sk_means.htm.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://portal.acm.org/citation.cfm?id=942806.943853.

[10] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in java workloads. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 270–287, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: http://doi.acm.org/10.1145/1028976.1028999. URL http://doi.acm.org/10.1145/1028976.1028999.

[11] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a jvm. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 24–34, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: http://doi.acm.org/10.1145/1356058.1356062. URL http://doi.acm.org/10.1145/1356058.1356062.

[12] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1186736.1186737.

[13] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 157–168, New York, NY, USA, 2003. ACM. ISBN 0-7695-1945-8. doi: http://doi.acm.org/10.1145/859618.859637. URL http://doi.acm.org/10.1145/859618.859637.

[14] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 95–104, New York, NY, USA, 2006. ACM. ISBN 1-59593-264-X. doi: http://doi.acm.org/10.1145/1152154.1152172. URL http://doi.acm.org/10.1145/1152154.1152172.

[15] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, volume 3b: system programming guide edition, September 2010. 30.4.4 Precise Event Based Sampling (PEBS).

[16] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO 39: Proceedings*

of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: http://dx.doi.org/10.1109/MICRO.2006.30.

[17] J. Kim, S. V. K. W. chung Hsu, D. J. Lilja, and P. chung Yew. Dynamic code region (dcr)-based program phase tracking and prediction for dynamic optimizations. In *in International Conference on High Performance Embedded Architectures and Compilers*, page 2005. Springer Verlag, 2005.

[18] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 57–67, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8385-0.

[19] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 236–247, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-8965-4. doi: http://dx.doi.org/10.1109/ISPASS.2005.1430578.

[20] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 278–289, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: http://dx.doi.org/10.1109/HPCA.2005.39.

[21] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. Technical Report Version 1.0, Intel Corporation, 2009.

[22] Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre, and A. Tarkas. Dynamic performance tuning for speculative threads. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 462–473, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555812. URL `http://doi.acm.org/10.1145/1555754.1555812`.

[23] D. MacKay. *Information Theory, Inference and Learning Algorithms*, chapter Chapter 20. An Example Inference Task: Clustering, pages 284–292. Cambridge University Press, 2003. ISBN 0-521-64298-1.

[24] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 191–202, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: http://dx.doi.org/10.1109/CGO.2005.26. URL `http://dx.doi.org/10.1109/CGO.2005.26`.

[25] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *CGO '06: Proceedings of the International Symposium on Code Generation*

and Optimization, pages 111–123, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: http://dx.doi.org/10.1109/CGO.2006.26.

[26] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 150–162, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: http://dx.doi.org/10.1109/PACT.2007.25. URL `http://dx.doi.org/10.1109/PACT.2007.25`.

[27] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '05, pages 321–326, New York, NY, USA, 2005. ACM. ISBN 1-59593-161-9. doi: http://doi.acm.org/10.1145/1084834.1084913. URL `http://doi.acm.org/10.1145/1084834.1084913`.

[28] E. Perelman, M. Polito, J. yves Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *In International Parallel and Distributed Processing Symposium*, pages 25–29, 2006.

[29] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 165–176, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: http://doi.acm.org/10.1145/1024393.1024414. URL `http://doi.acm.org/10.1145/1024393.1024414`.

[30] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. Technical report, La Jolla, CA, USA, 2001.

[31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 45–57, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. doi: http://doi.acm.org/10.1145/605397.605403. URL `http://doi.acm.org/10.1145/605397.605403`.

[32] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *SIGARCH Comput. Archit. News*, 31(2):336–349, 2003. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/871656.859657.

[33] T. Sondag and H. Rajan. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 73–80, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3718-4. doi: http://dx.doi.org/10.1109/IWMSE.2009.5071386.

[34] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. *WBIA Workshop at ASPLOS*, 2006.

A libloopppac

libloopppac is a low overhead online phase prediction and classification library written in C++. The interface exposes a set of C functions in a single header file, see Listings B. libloopppac requires the library user to register a signal handler for SIGIO and forward any signal to the library with *looppac_forward_signal*. The library then check the destination, collect sample data from Linux perf_events if the signal was destined to the library.

The following steps show how to use the library.

1. Initialize the library with *looppac_init*. The function takes one argument, the name of the file where the log messages will be written to.
2. Monitoring is done at thread level. The function *looppac_open_monitor* opens a monitor and returns a monitor handle. It takes two arguments, the thread id and a data structure, *looppac_monitor_attr*, with all the monitor settings.
3. Start the monitor with *looppac_start_monitor*.
4. Forward any signal to the library with *looppac_forward_signal*. The library has two callback functions, one when for every interval and one when the phase changes. A data structure , *looppac_event_info*, that contains among other things the phase id and a prediction.
5. Stop the monitor with *looppac_stop_monitor*.
6. The function *looppac_close_monitor* closes the monitor and free all resources.
7. Shutdown the library with *looppac_shutdown*.

To use all the features such as randomized sampling and precise event based sampling for branch instruction, the Linux kernel must be patched. The library will still perform well without these features but at a lower accuracy, i.e. larger intra-phase variance or a increase in number of detected phases.

B SPEC2006 Signatures

The following section shows how the CPI and the signature various over time for all the SPEC applications. All the application were run from start to completion. 100 million instructions intervals were used with a sample period of 1 thousand branch instructions. For example, Figure B.1 shows the whole execution of *astar/lakes*. It has very visible phases for both CPI and signature, a phase change happens after 50 billion instructions (500 intervals) and the CPI jumps from 2 to 2.5.

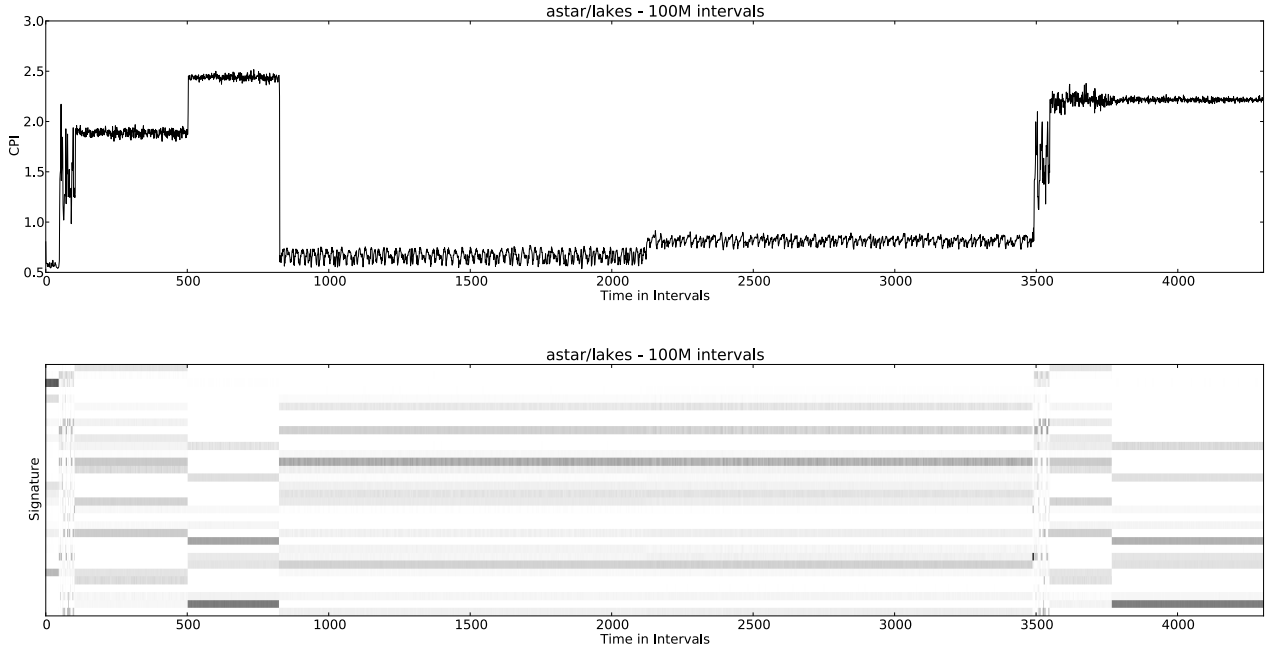


Figure B.1: CPI and the Signature for *astar/lakes*.

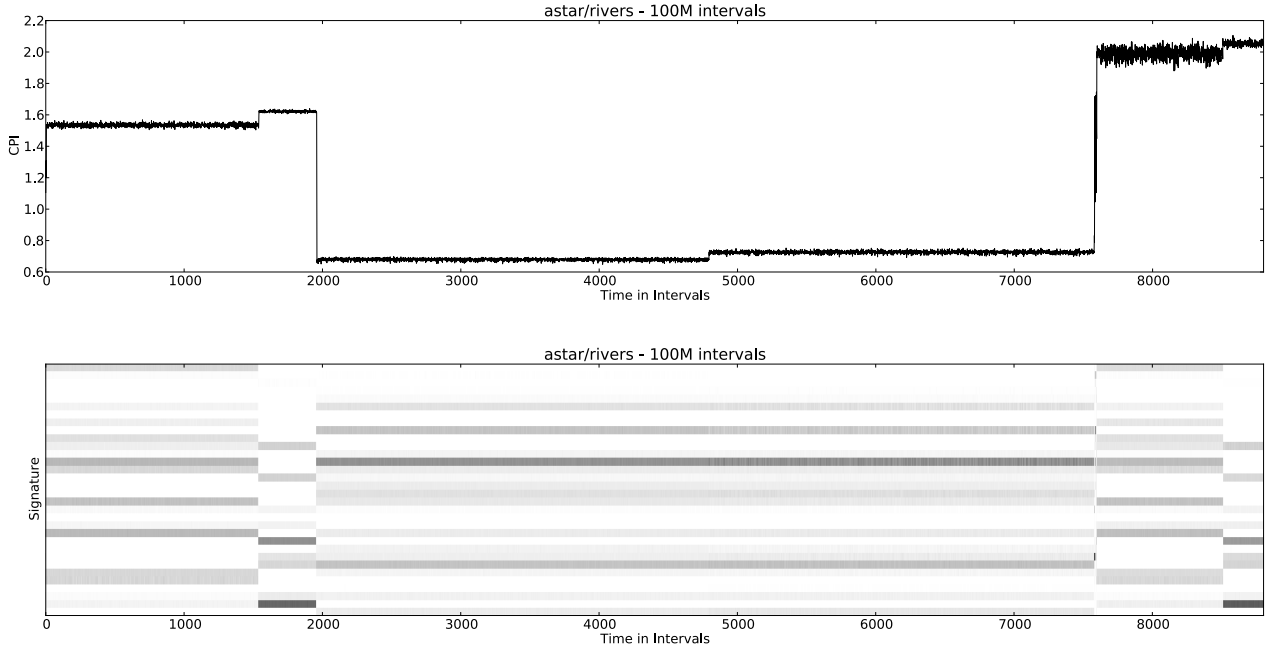


Figure B.2: CPI and the Signature for *astar/rivers*.

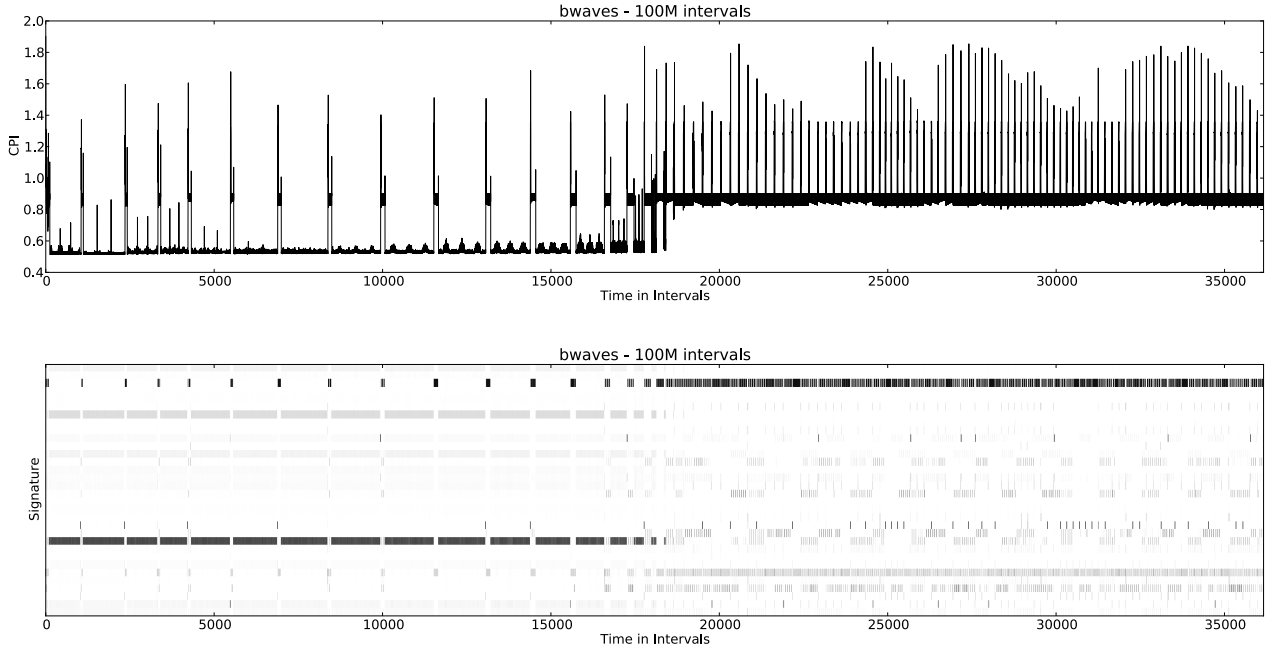


Figure B.3: CPI and the Signature for *bwaves*.

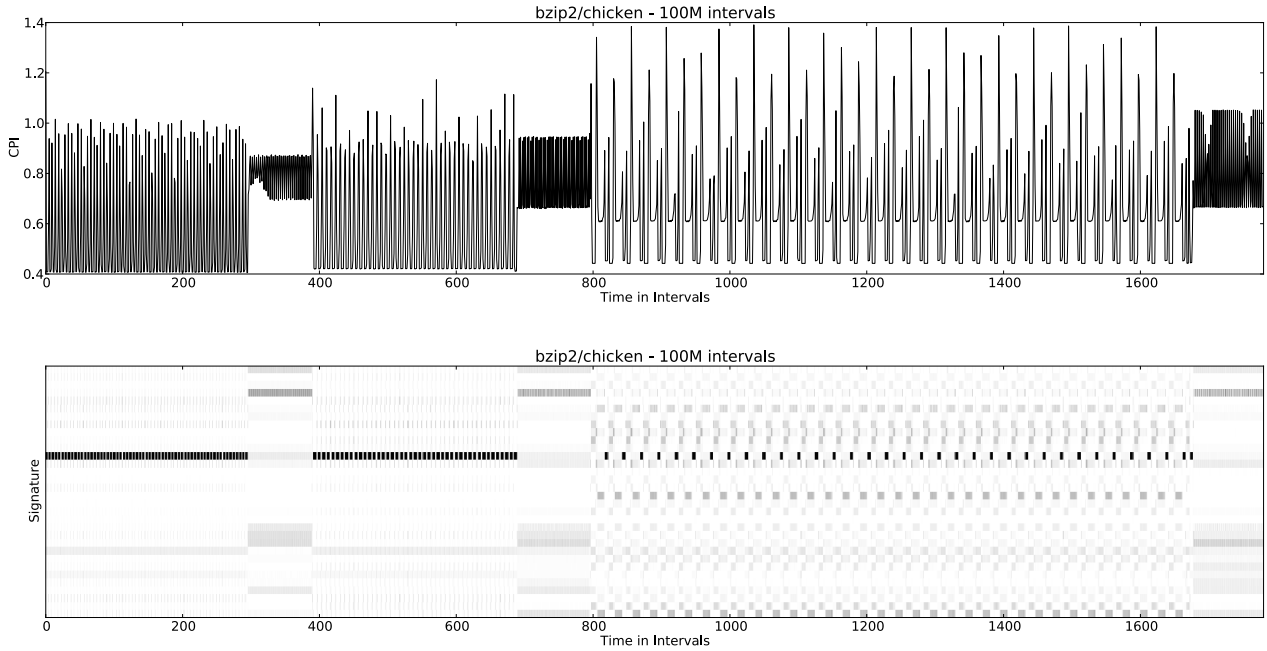


Figure B.4: CPI and the Signature for *bzip2/chicken*.

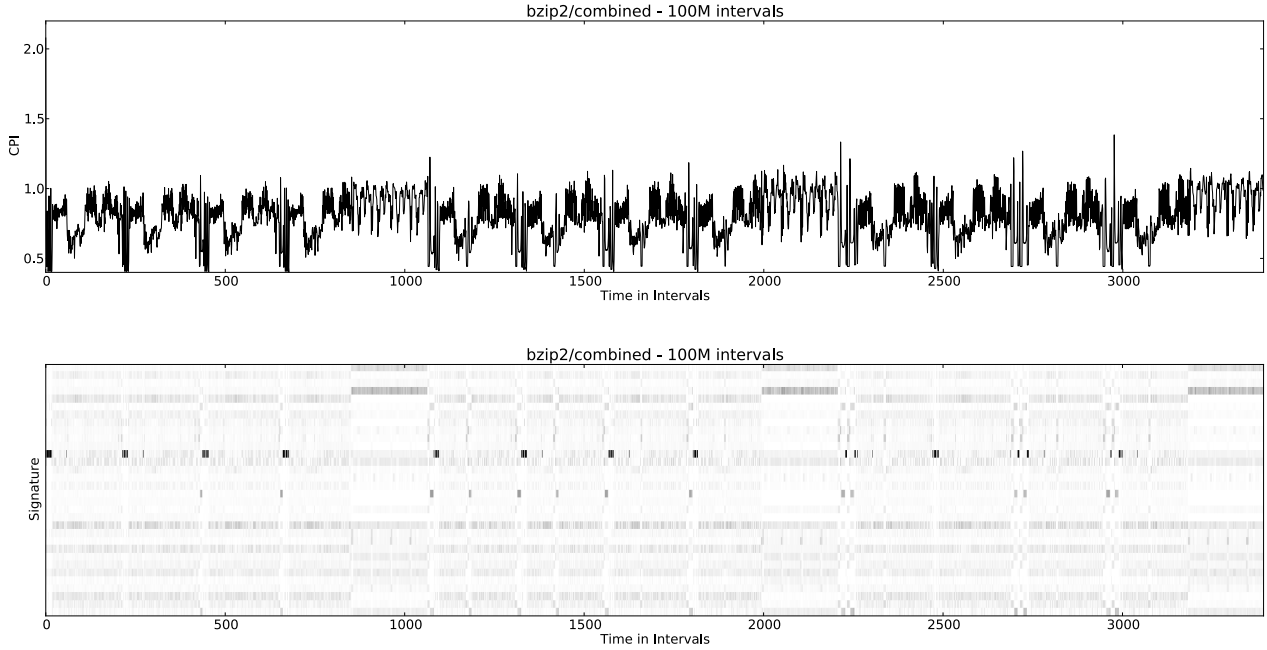


Figure B.5: CPI and the Signature for *bzip2/combined*.

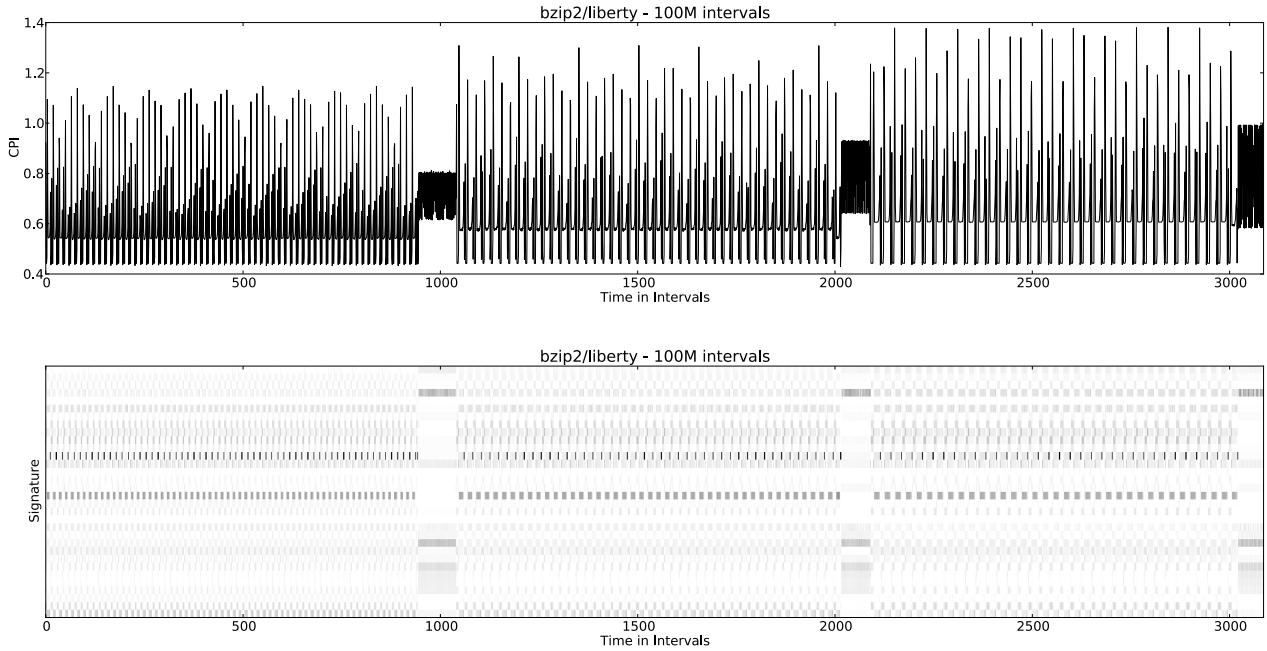


Figure B.6: CPI and the Signature for *bzip2/liberty*.

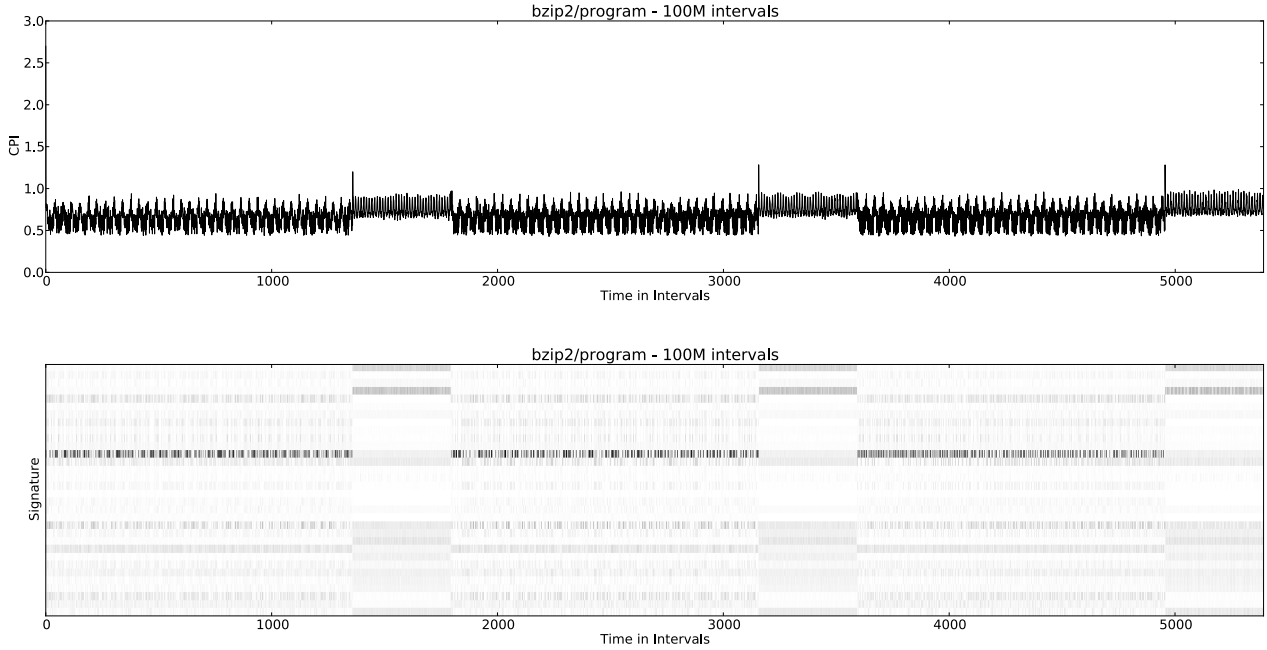


Figure B.7: CPI and the Signature for *bzip2/program*.

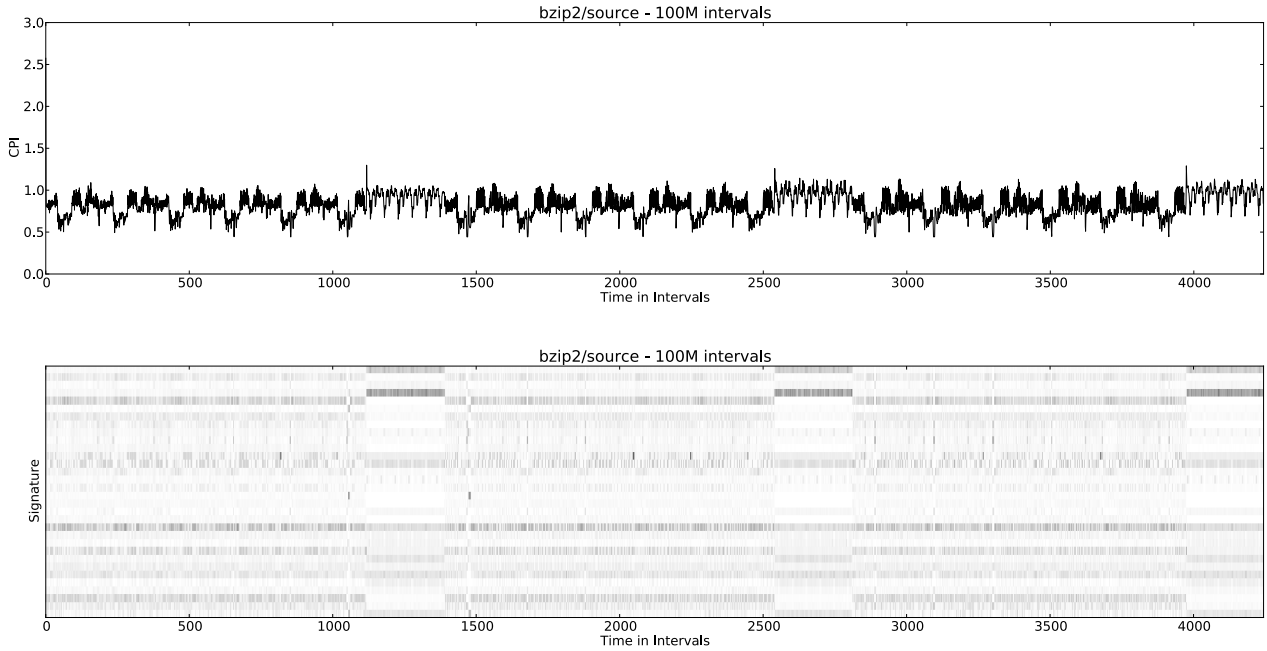


Figure B.8: CPI and the Signature for *bzip2/source*.

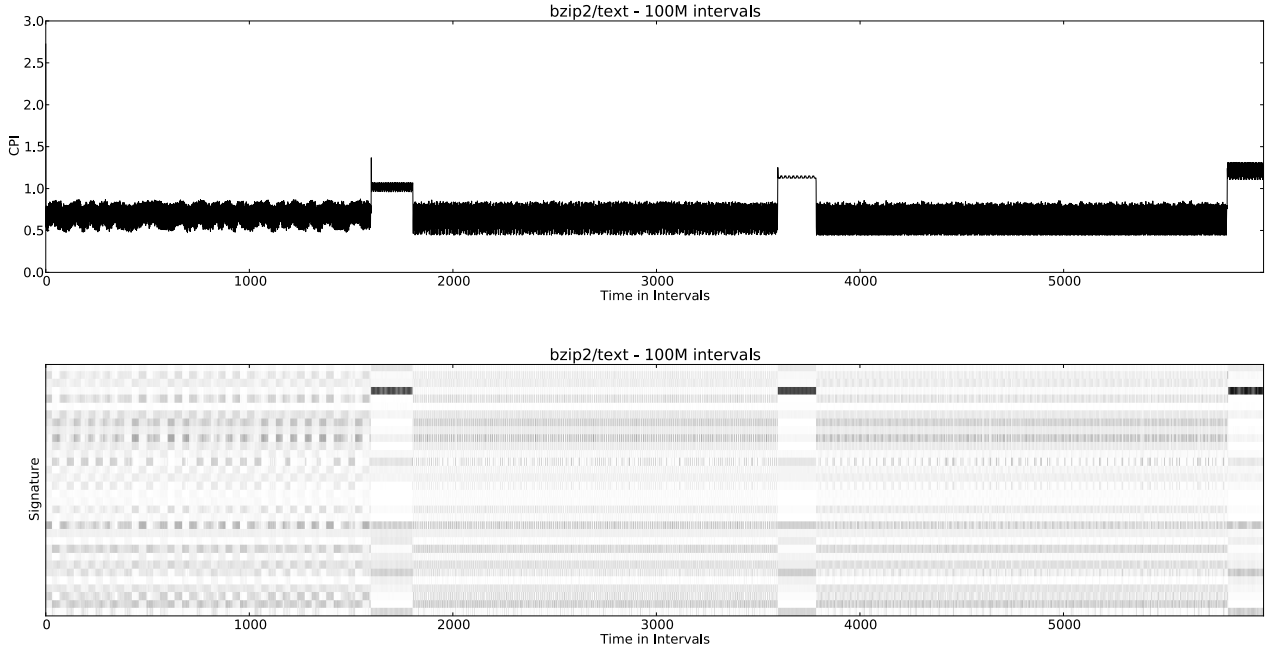


Figure B.9: CPI and the Signature for *bzip2/text*.

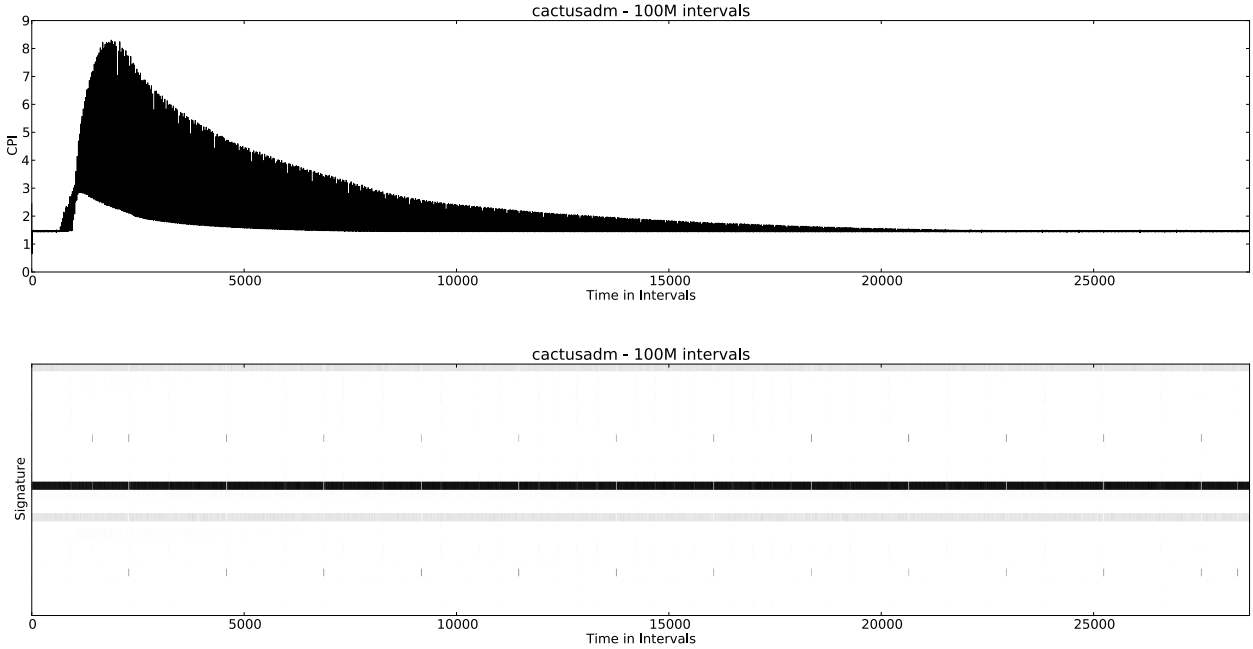


Figure B.10: CPI and the Signature for *cactusadm*.

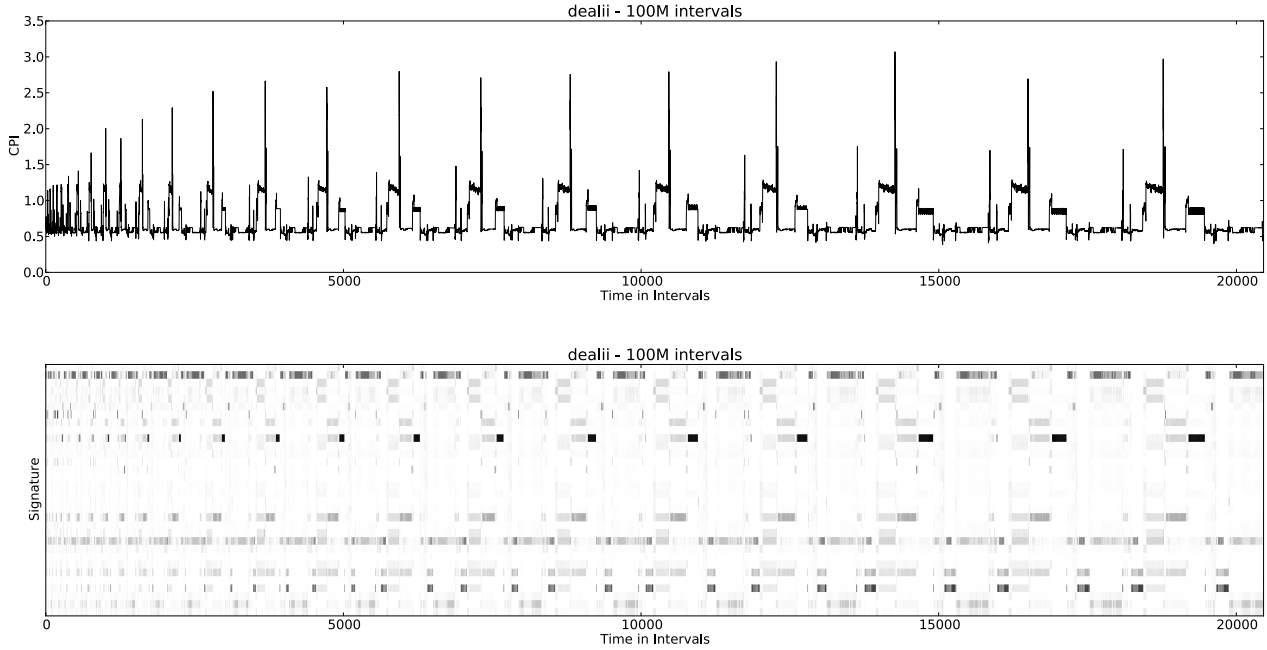


Figure B.11: CPI and the Signature for *dealii*.

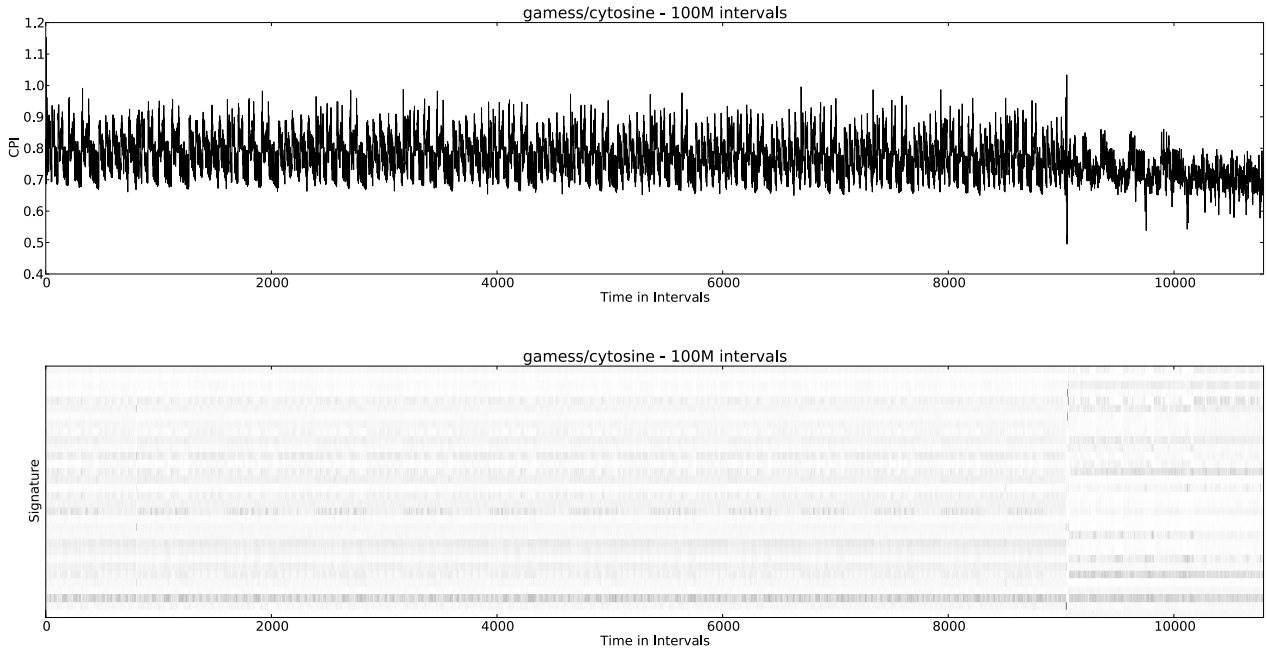


Figure B.12: CPI and the Signature for *gamess/cytosine*.

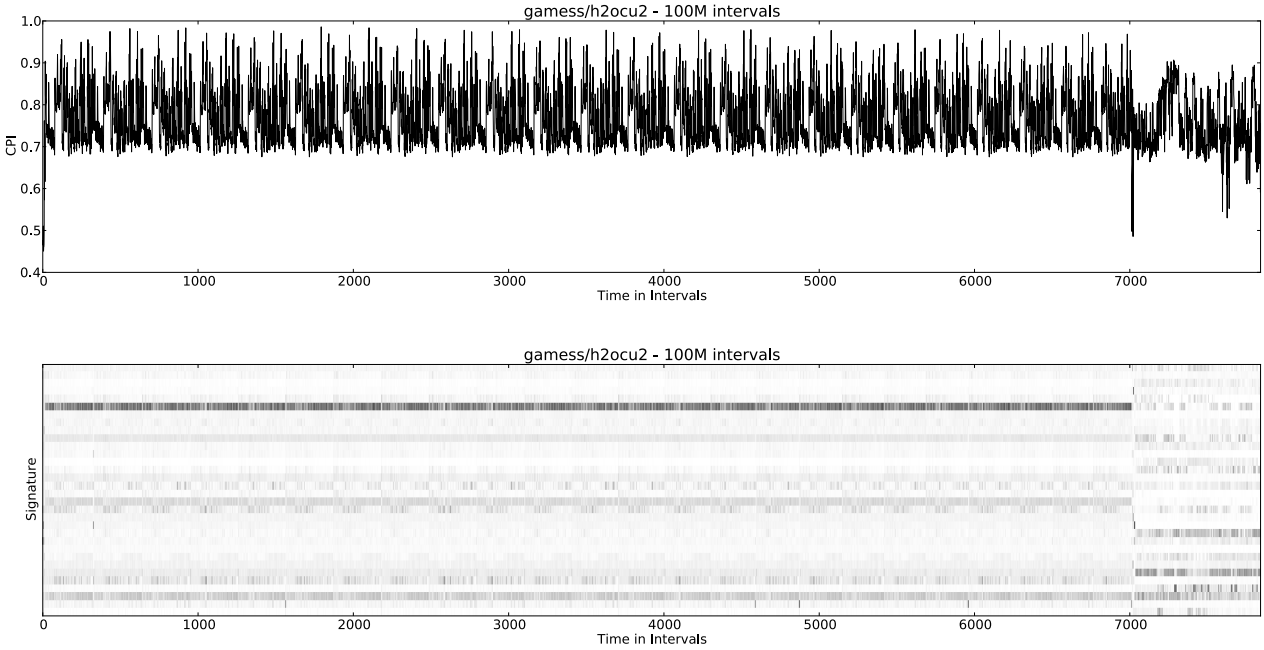


Figure B.13: CPI and the Signature for *gamess/h2ocu2*.

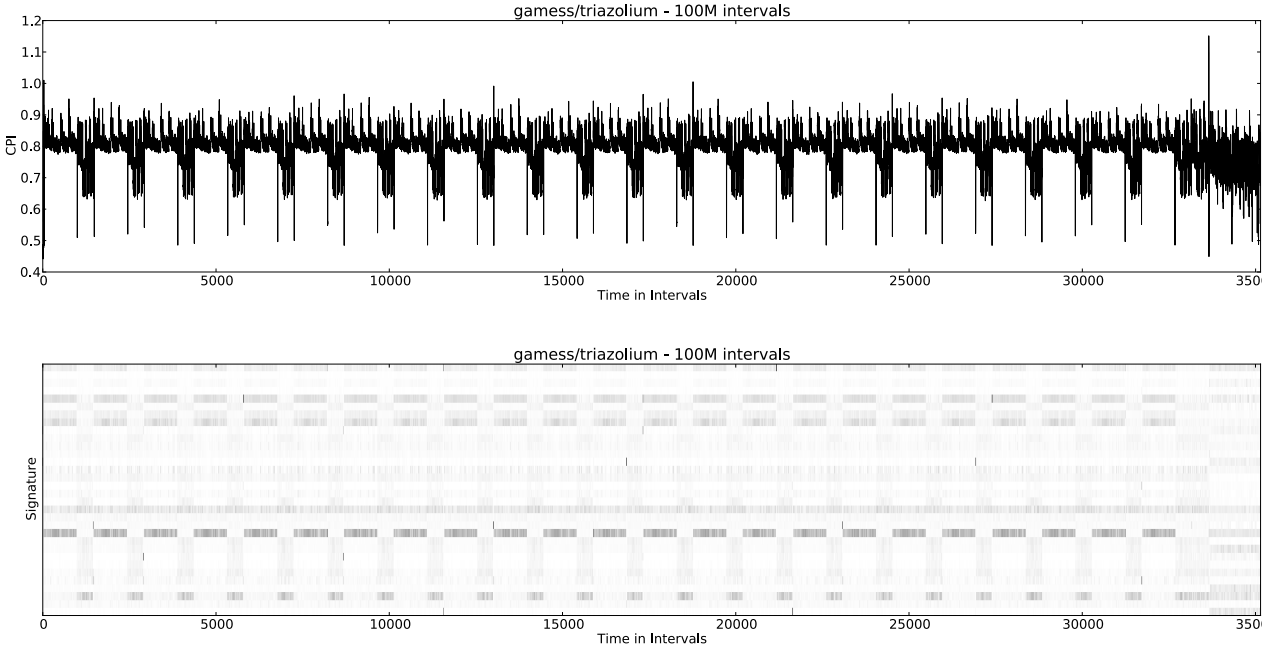


Figure B.14: CPI and the Signature for *gamess/triazolium*.

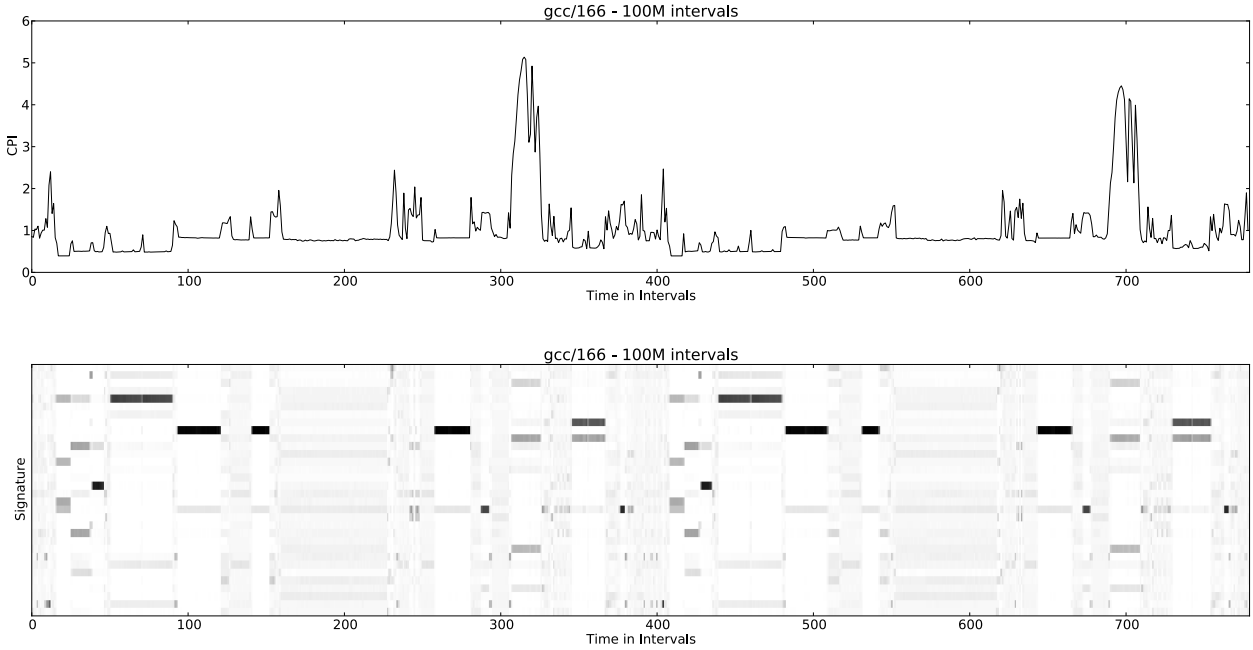


Figure B.15: CPI and the Signature for *gcc/166*.

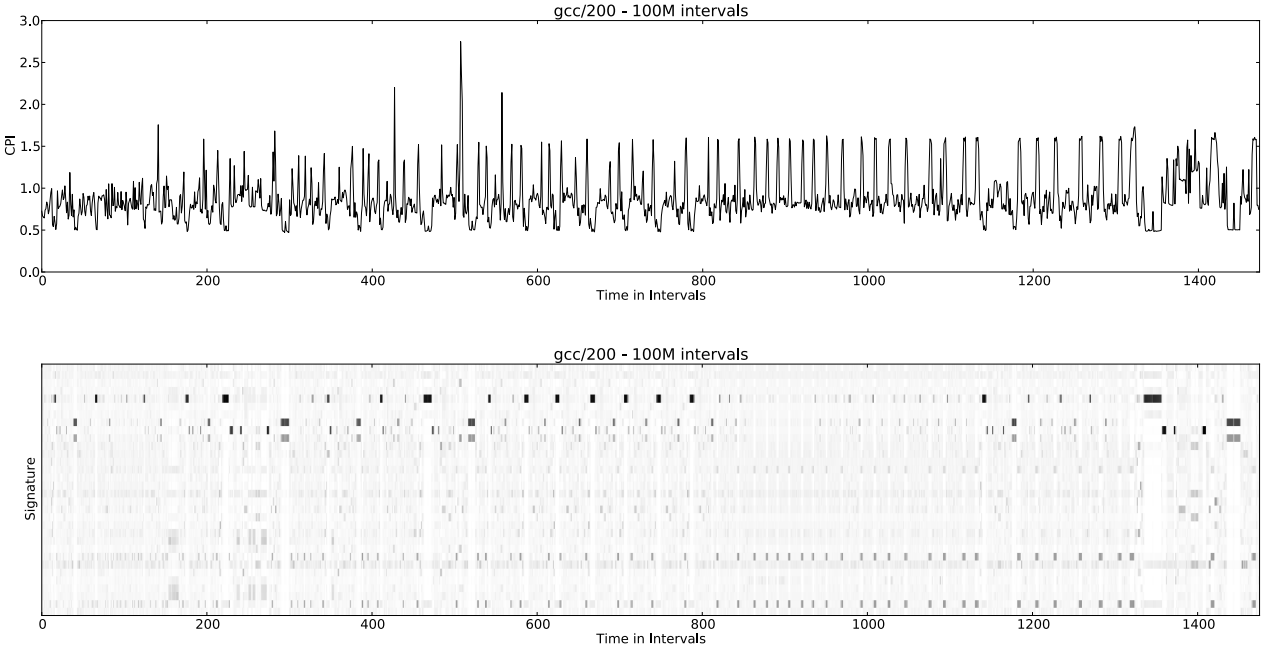


Figure B.16: CPI and the Signature for *gcc/200*.

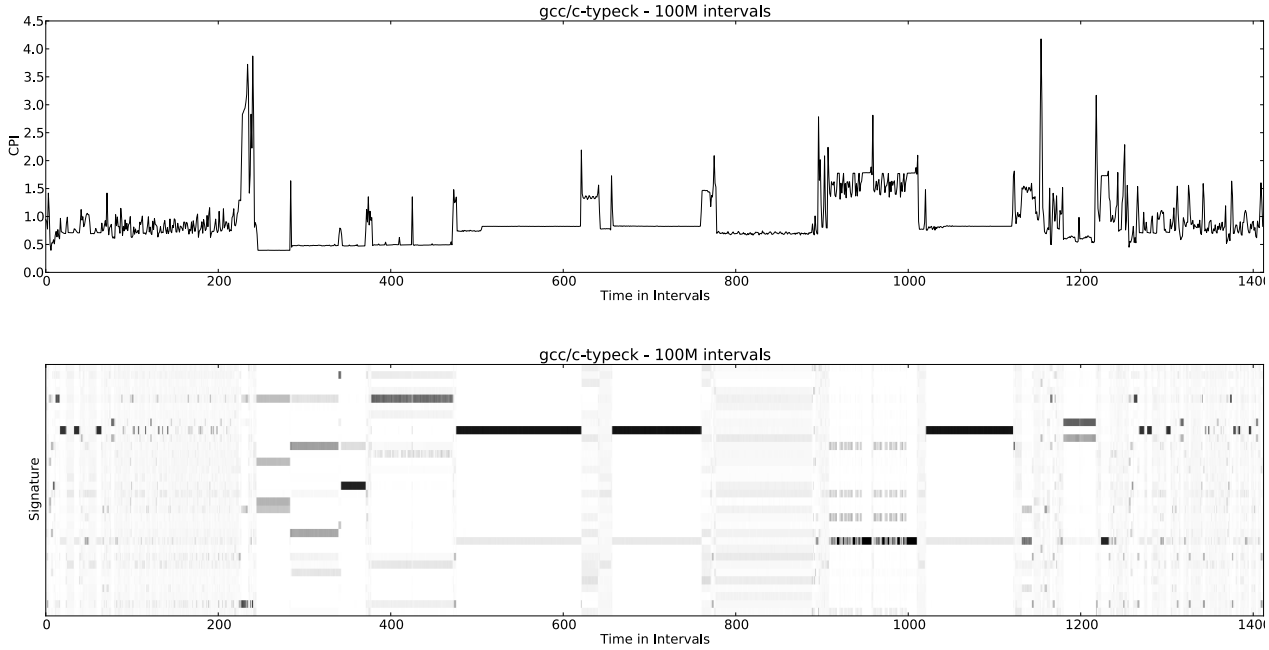


Figure B.17: CPI and the Signature for *gcc/c-typeck*.

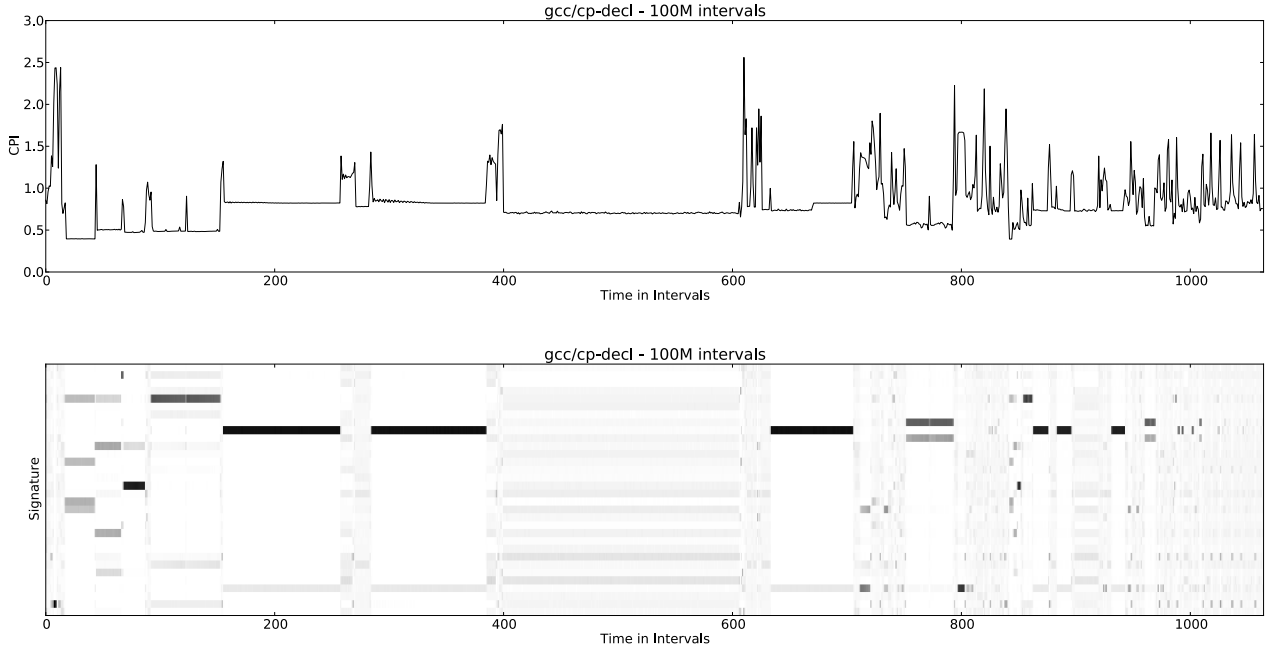


Figure B.18: CPI and the Signature for *gcc/cp-decl*.

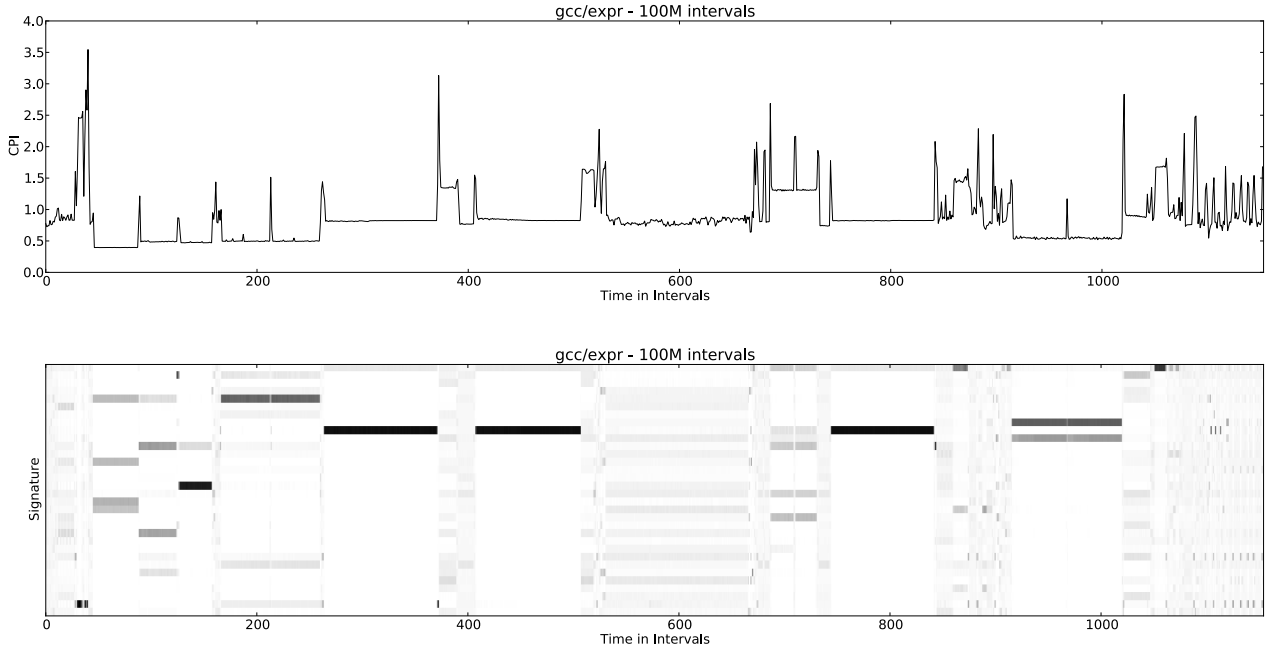


Figure B.19: CPI and the Signature for *gcc/expr*.

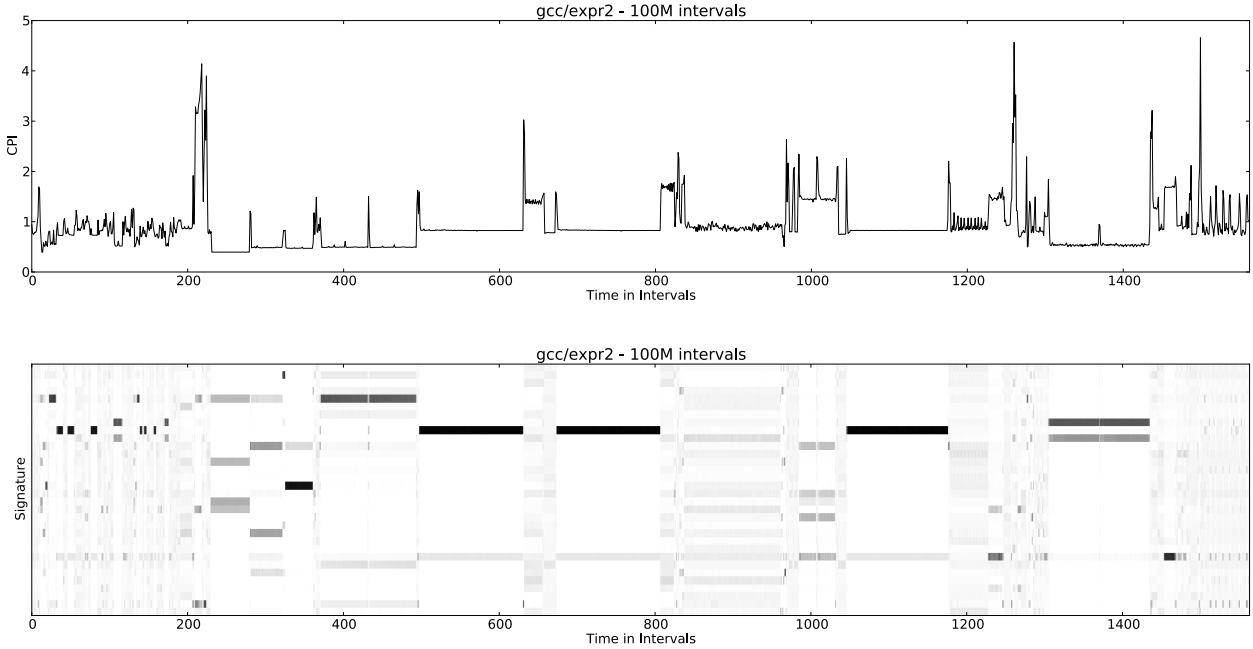


Figure B.20: CPI and the Signature for *gcc/expr2*.

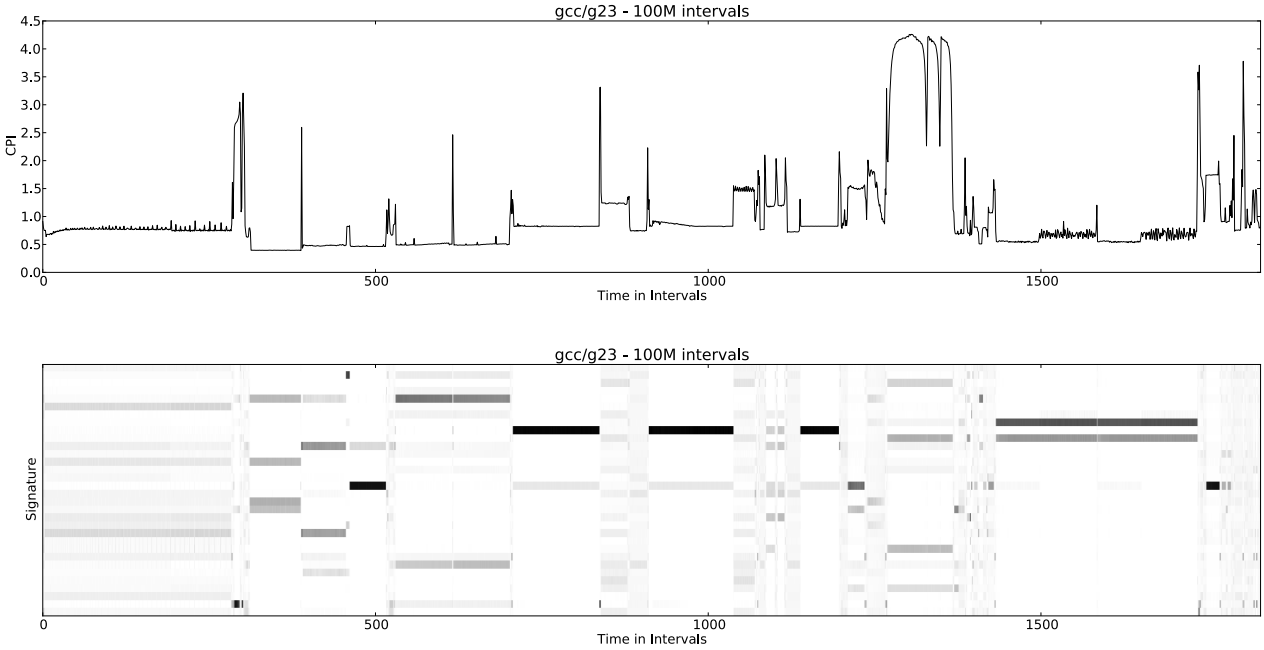


Figure B.21: CPI and the Signature for *gcc/g23*.

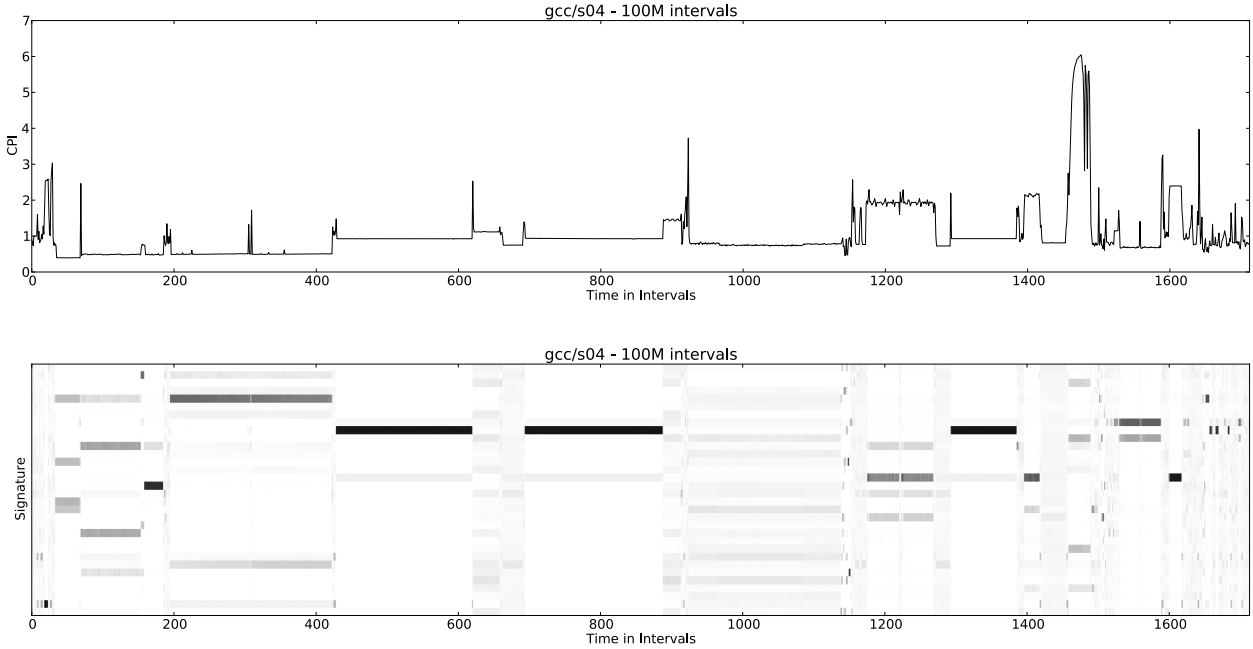


Figure B.22: CPI and the Signature for *gcc/s04*.

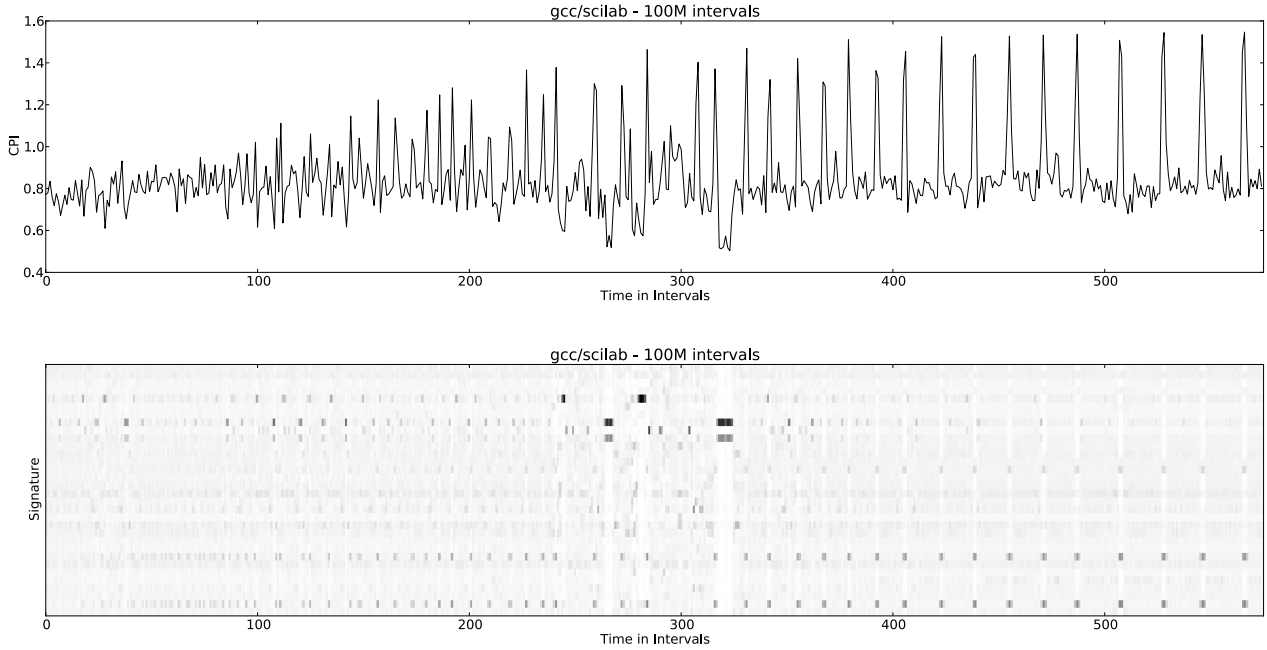


Figure B.23: CPI and the Signature for *gcc/scilab*.

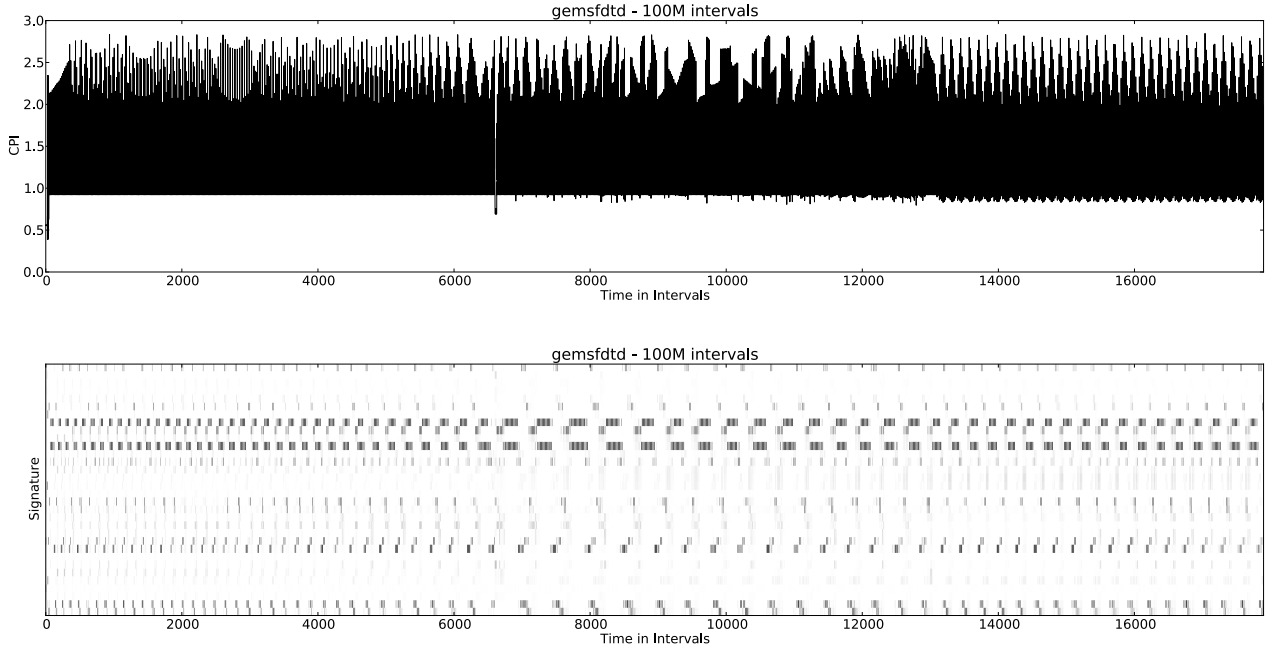


Figure B.24: CPI and the Signature for *gemsfdtd*.

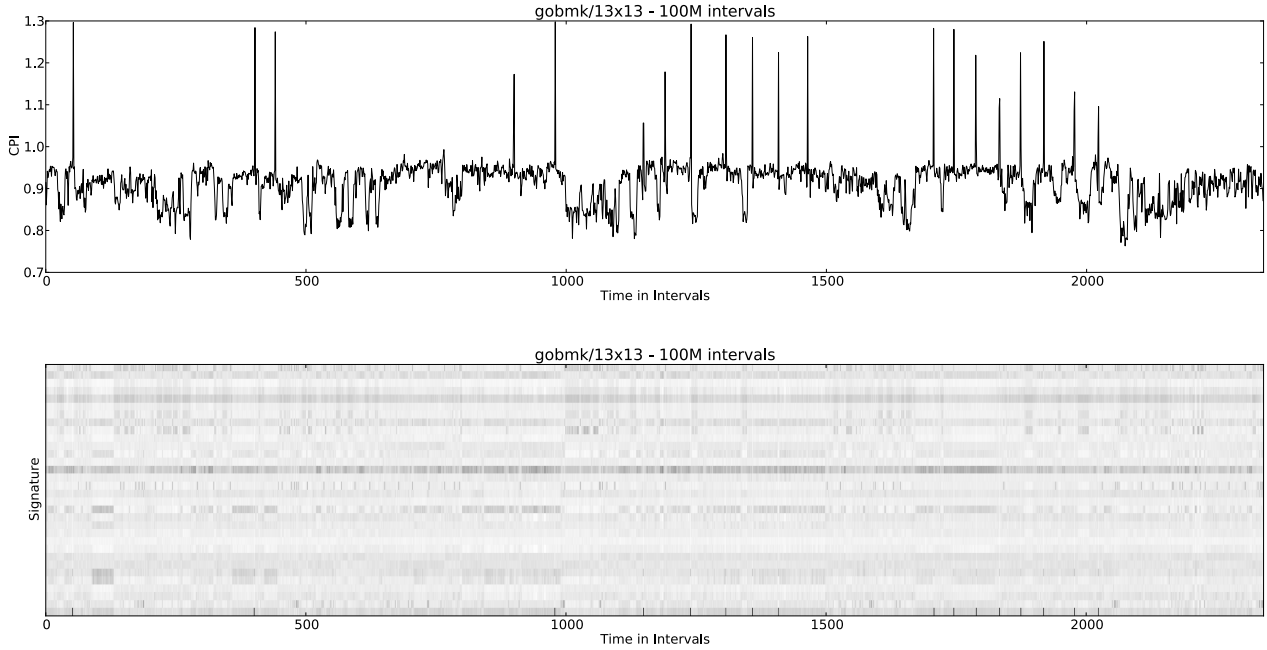


Figure B.25: CPI and the Signature for *gobmk/13x13*.

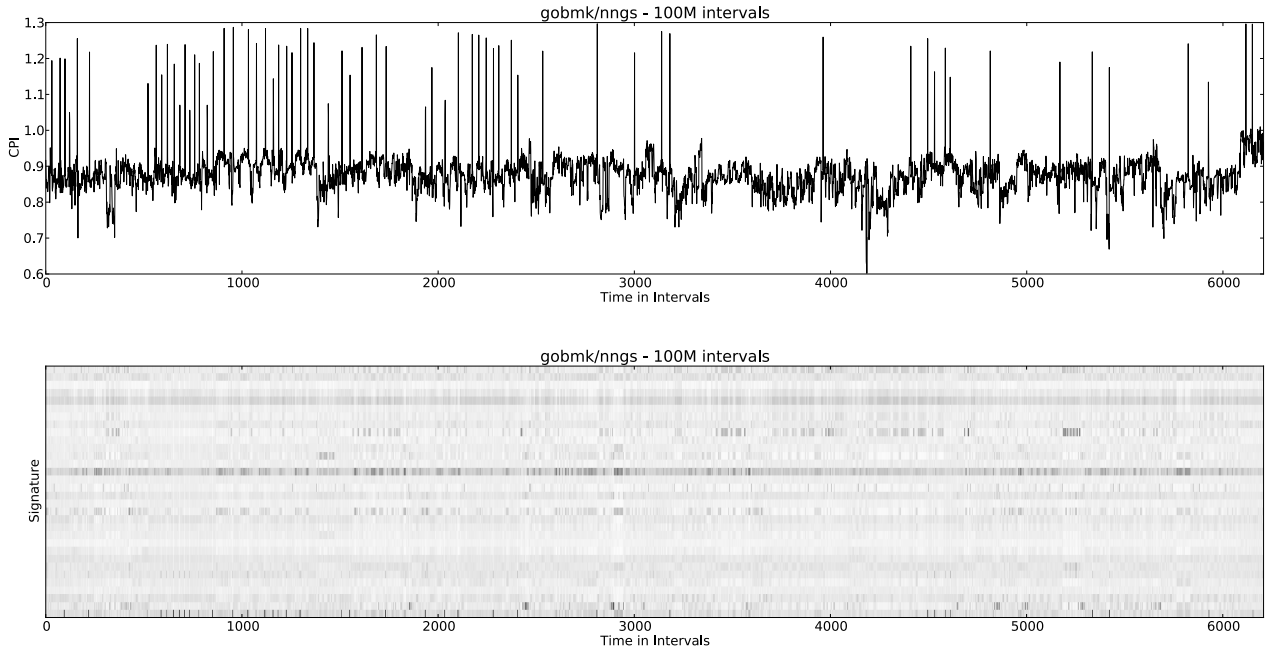


Figure B.26: CPI and the Signature for *gobmk/nngs*.

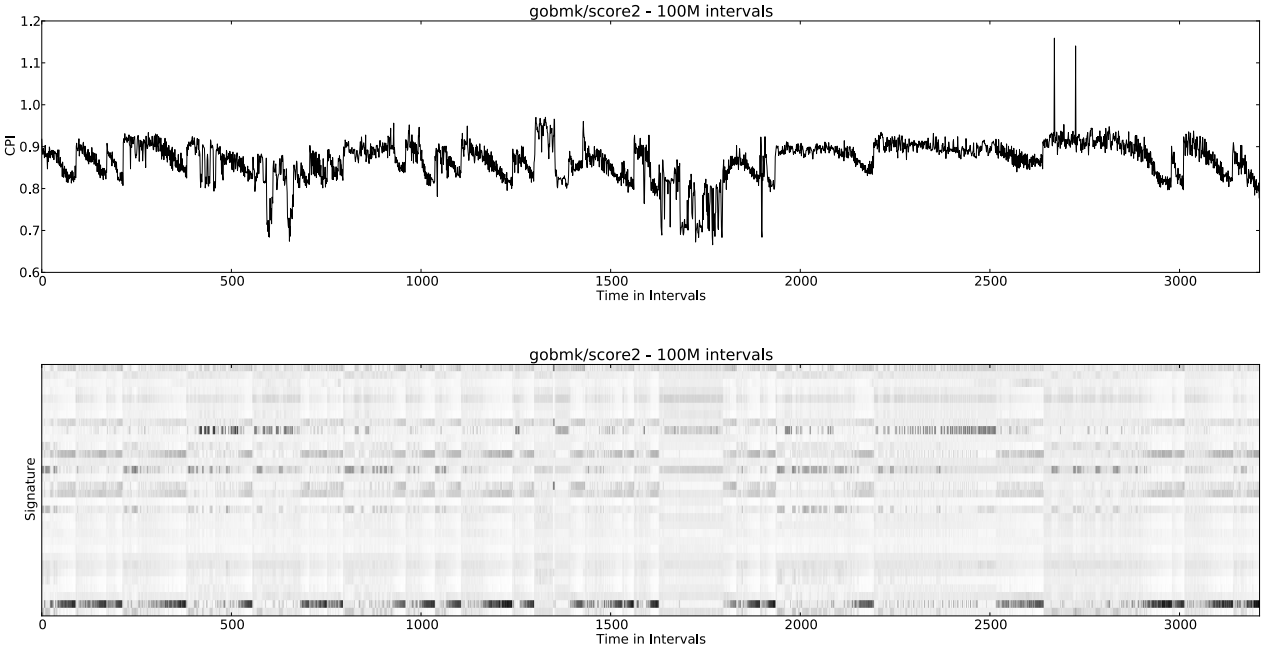


Figure B.27: CPI and the Signature for *gobmk/score2*.

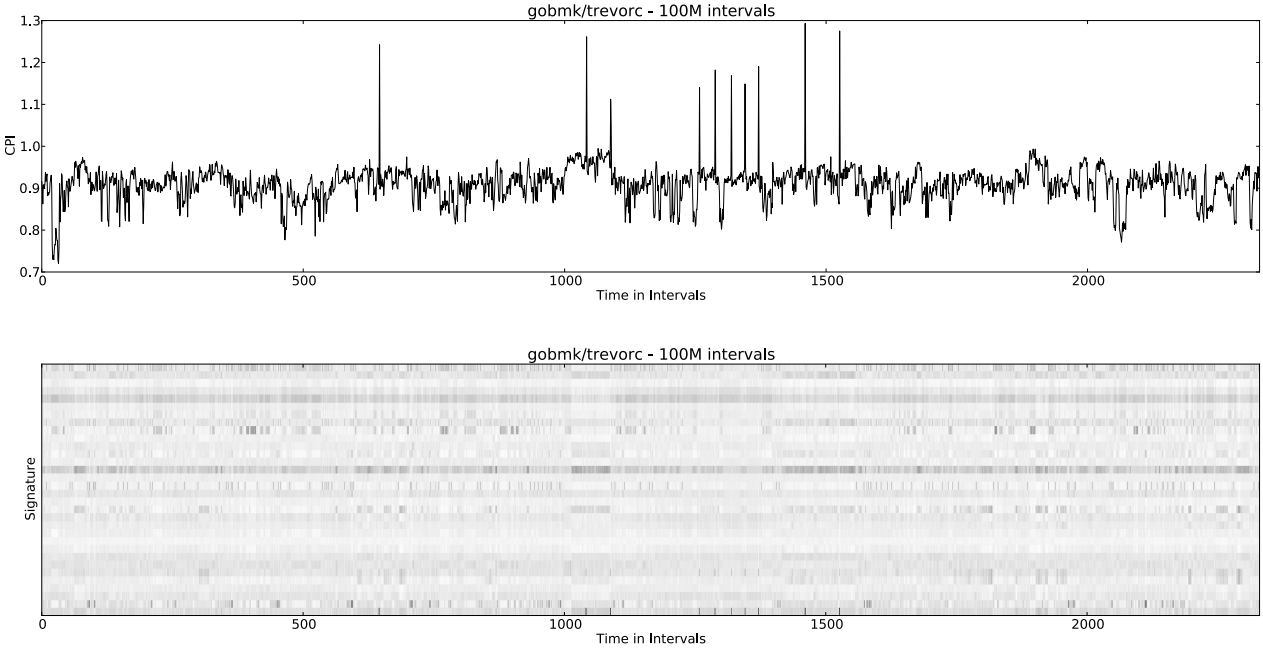


Figure B.28: CPI and the Signature for *gobmk/trevorc*.

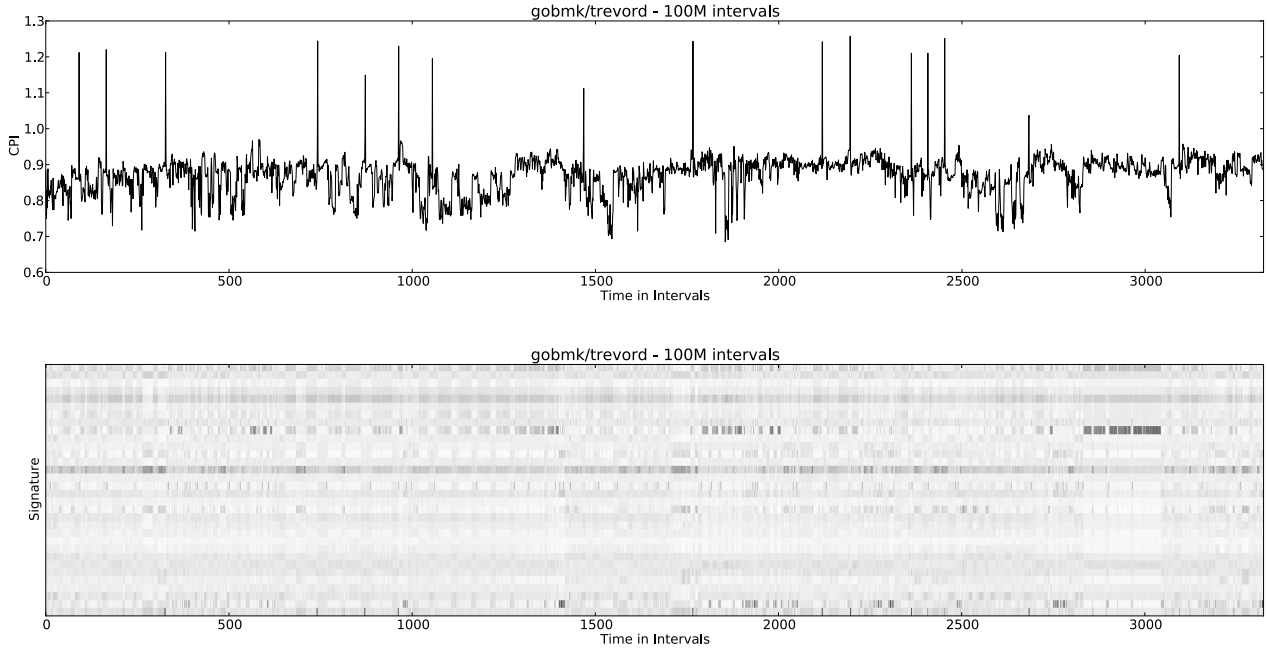


Figure B.29: CPI and the Signature for *gobmk/trevord*.

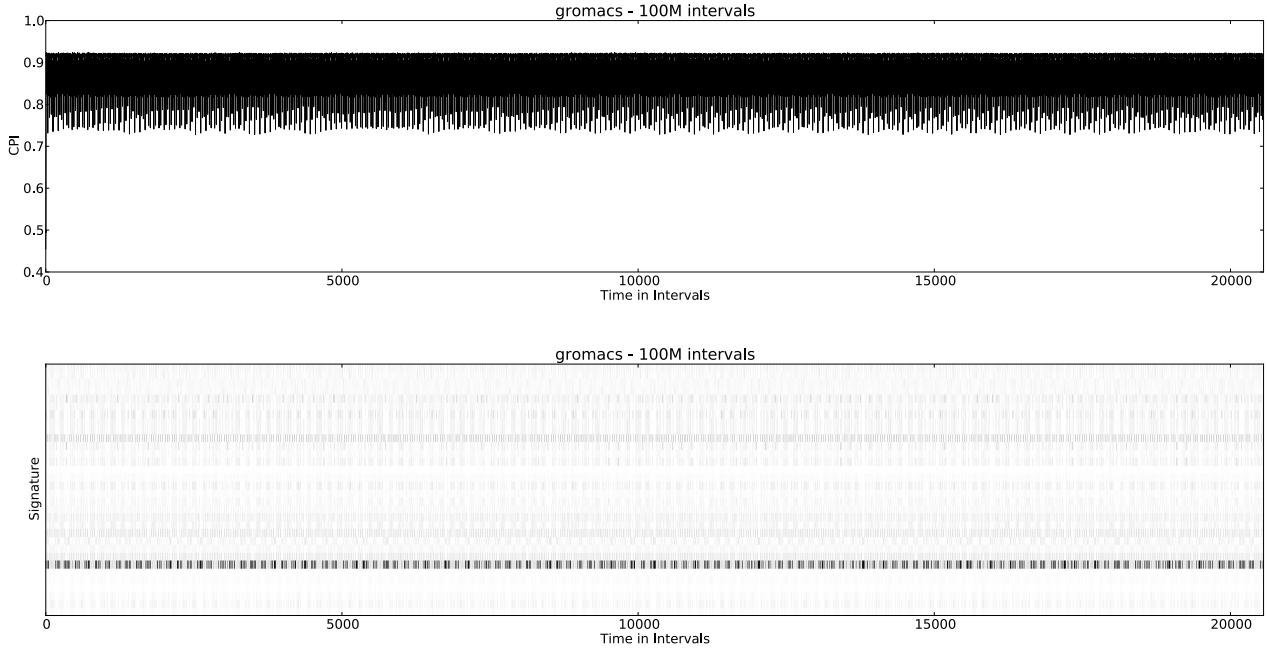


Figure B.30: CPI and the Signature for *gromacs*.

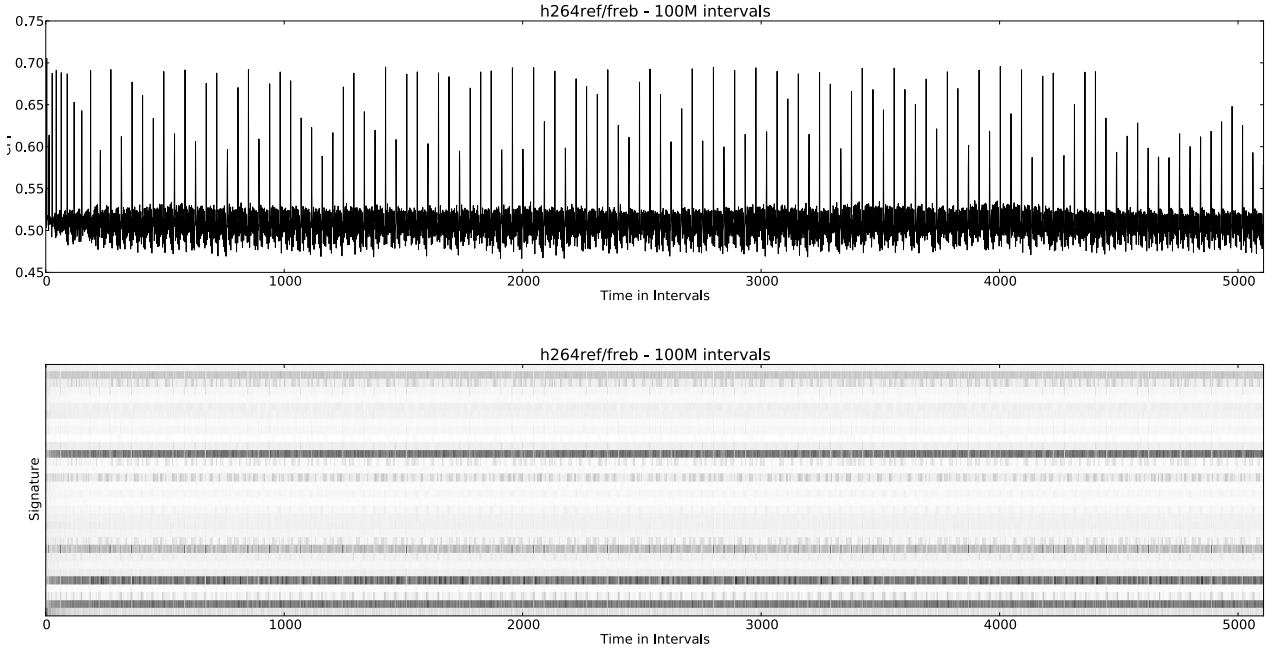


Figure B.31: CPI and the Signature for *h264ref/freb*.

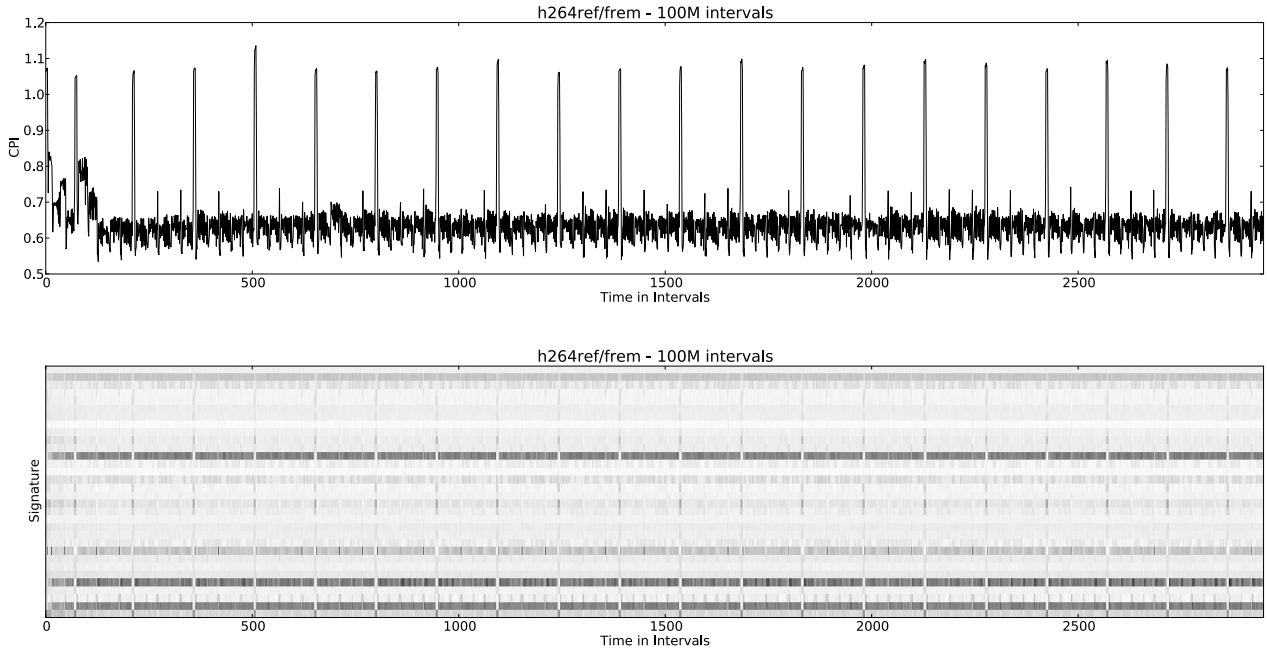


Figure B.32: CPI and the Signature for *h264ref/frem*.

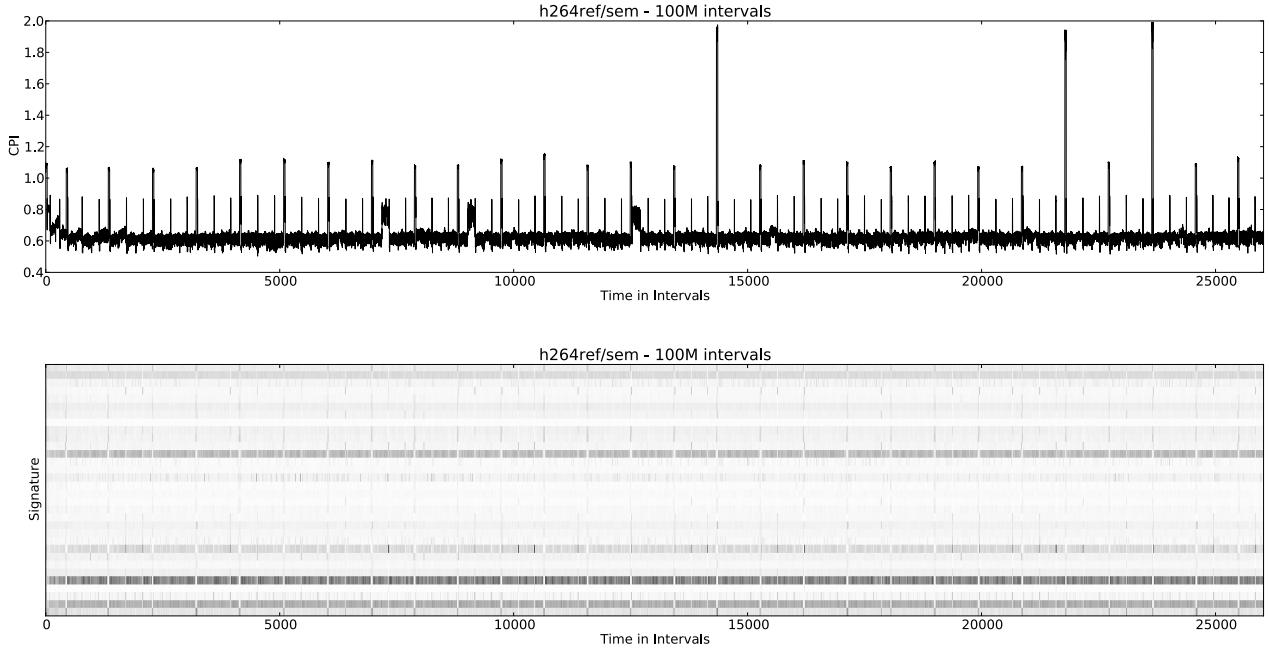


Figure B.33: CPI and the Signature for *h264ref/sem*.

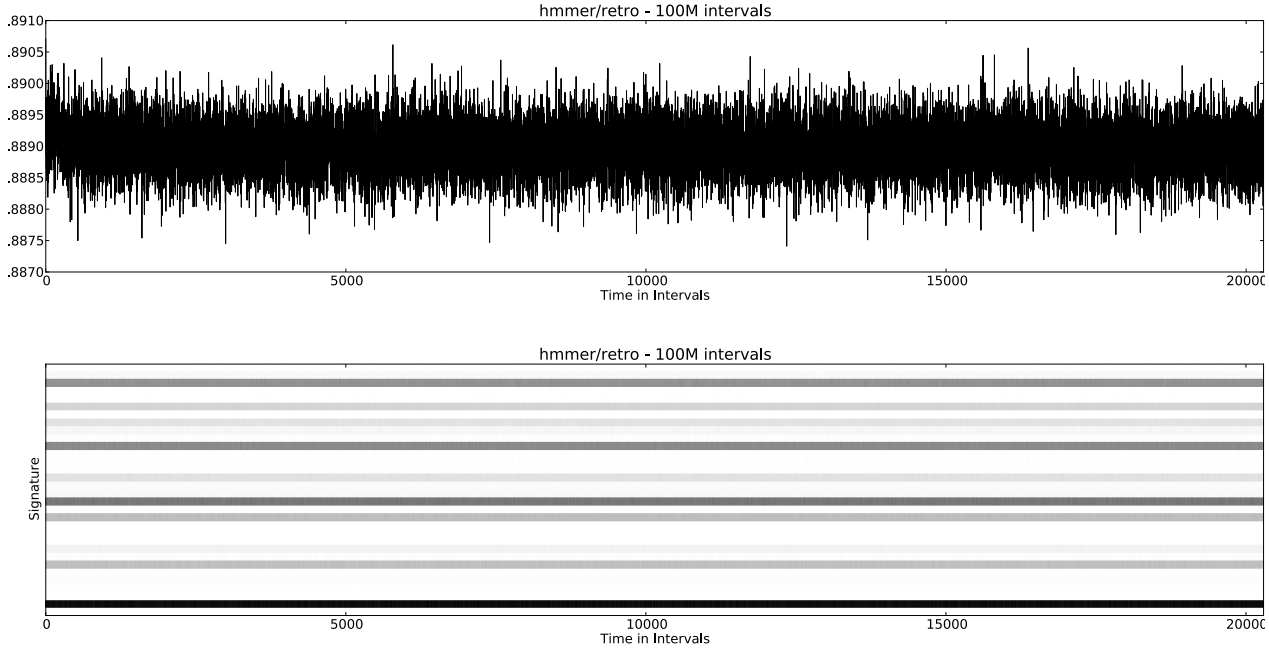


Figure B.34: CPI and the Signature for *hmmer/retro*.

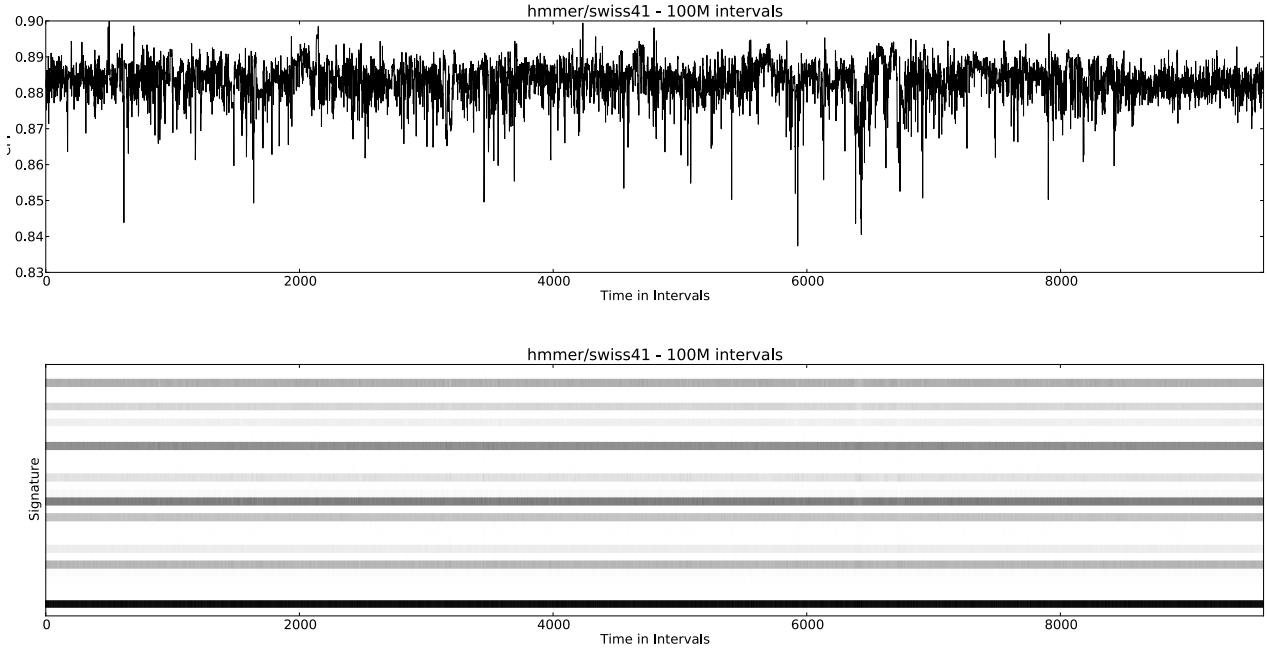


Figure B.35: CPI and the Signature for *hmmer/swiss41*.

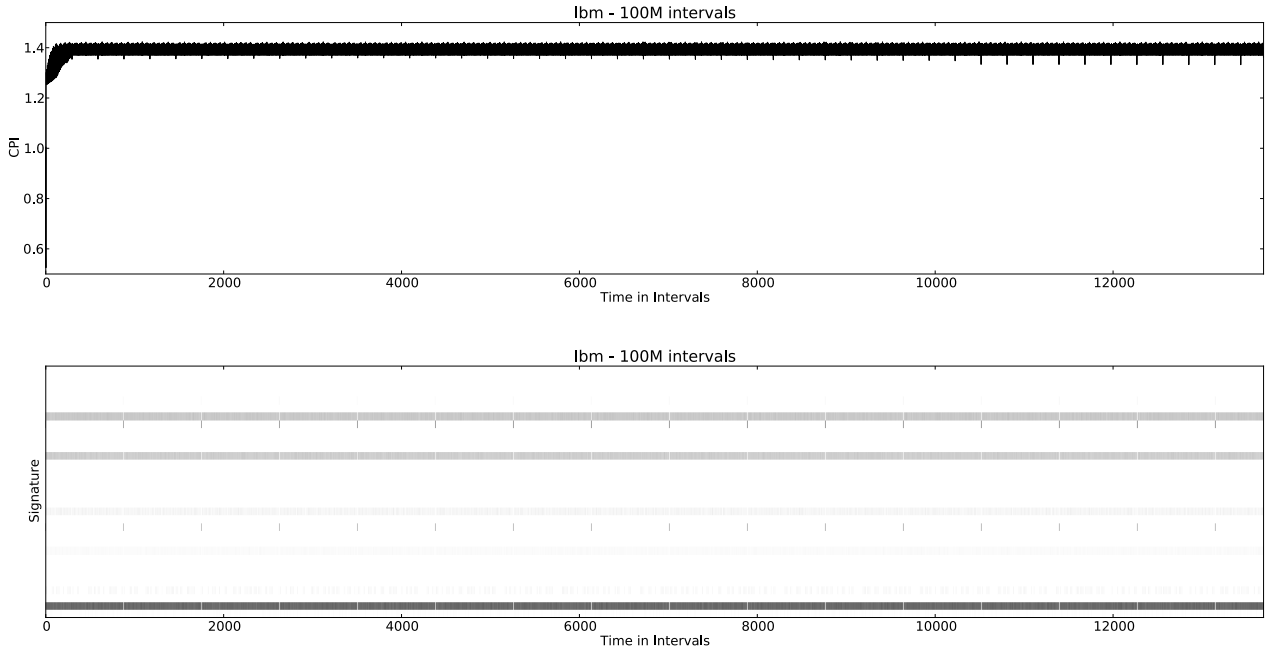


Figure B.36: CPI and the Signature for *lbm*.

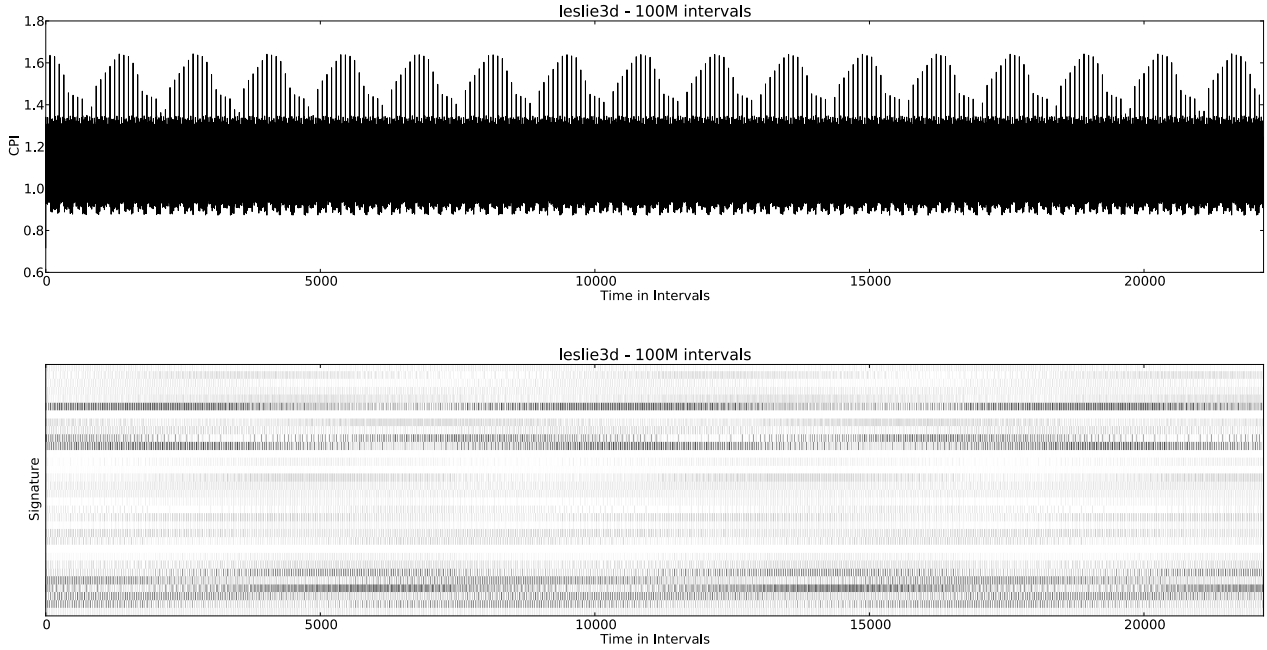


Figure B.37: CPI and the Signature for *leslie3d*.

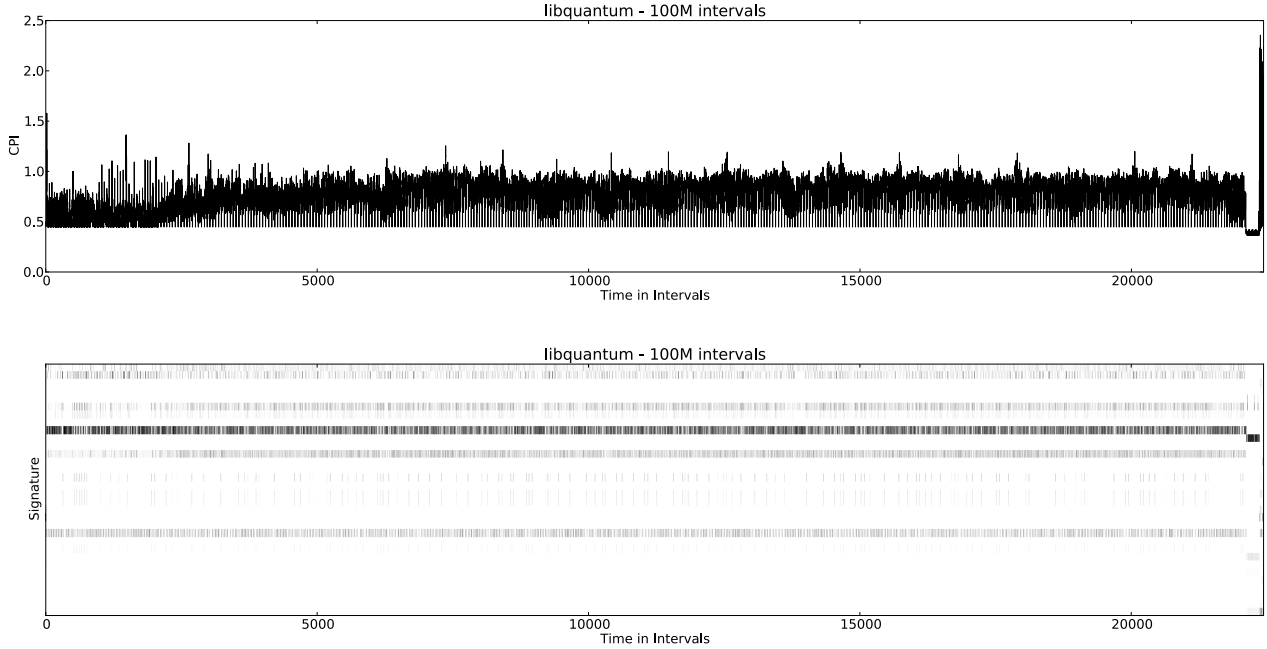


Figure B.38: CPI and the Signature for *libquantum*.

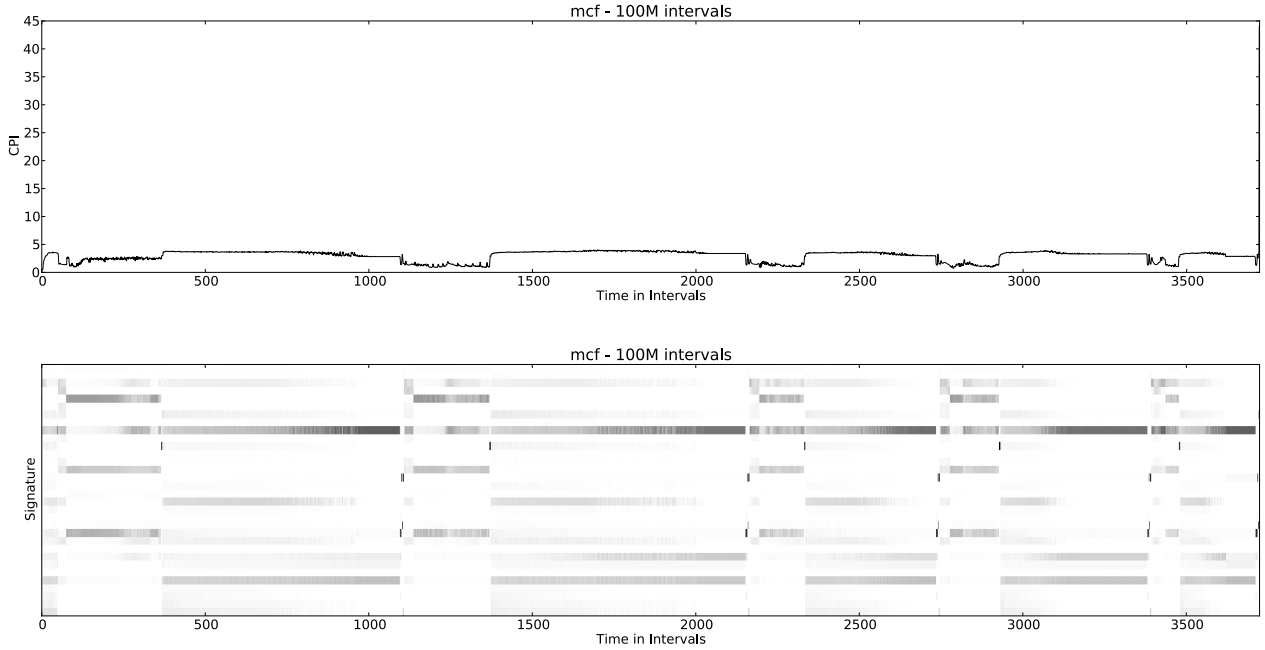


Figure B.39: CPI and the Signature for *mcf*.

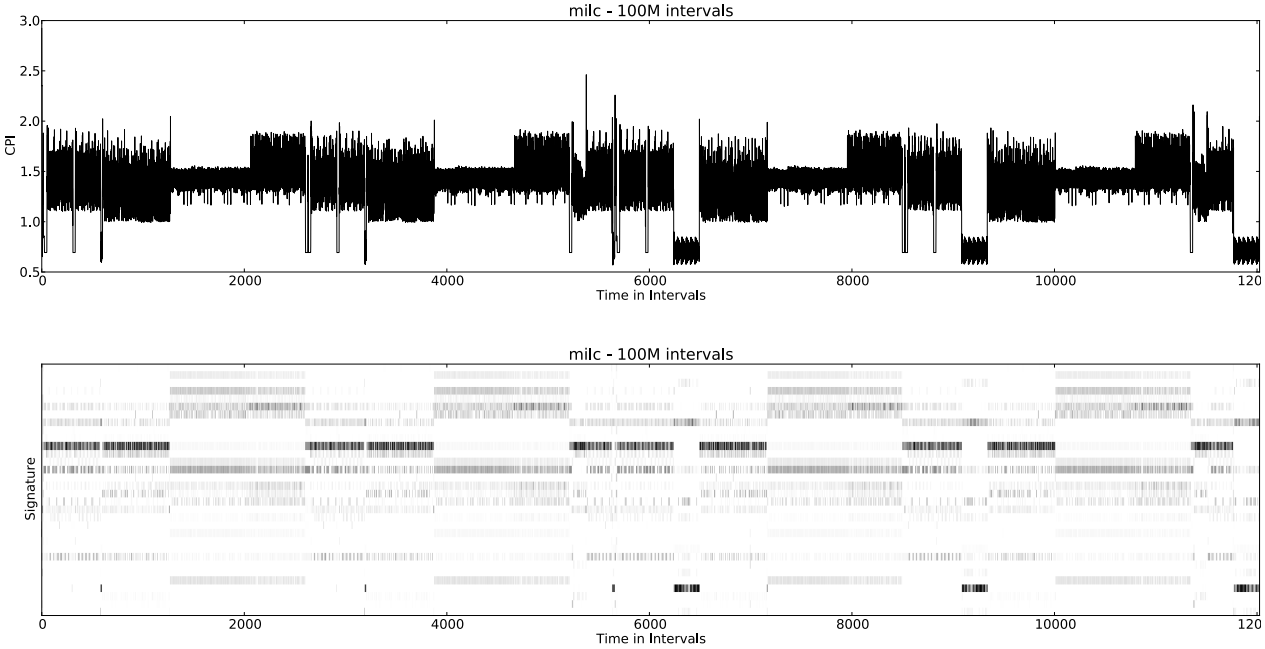


Figure B.40: CPI and the Signature for *milc*.

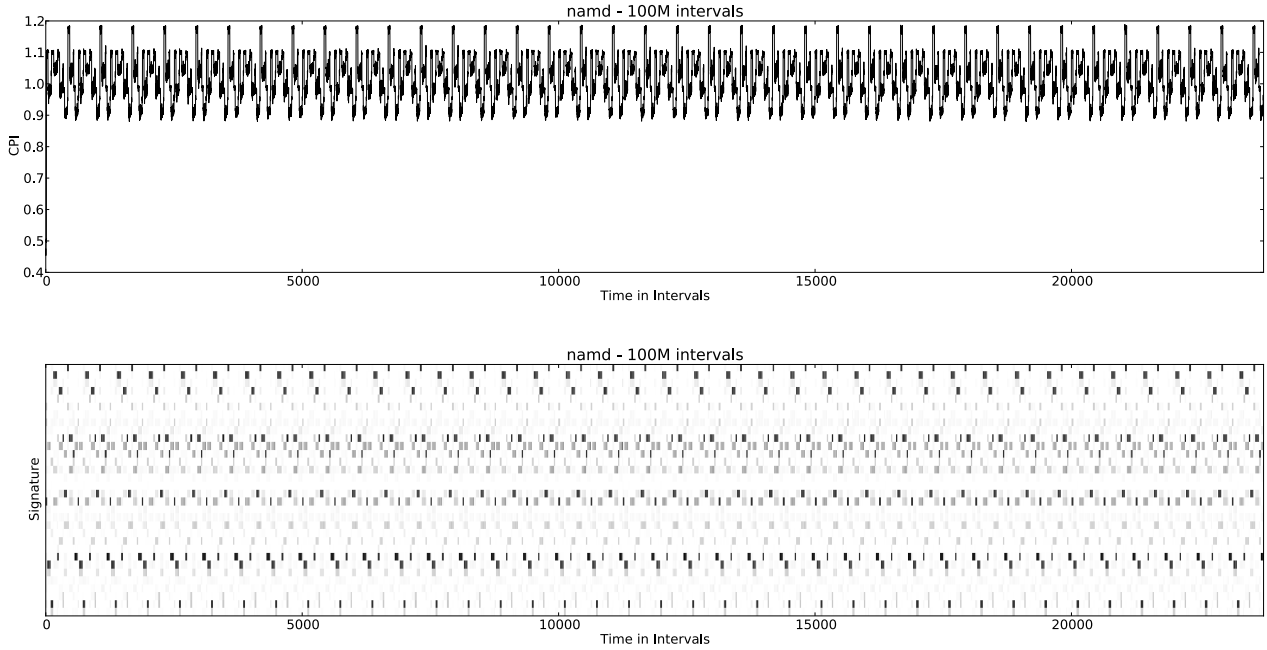


Figure B.41: CPI and the Signature for *namd*.

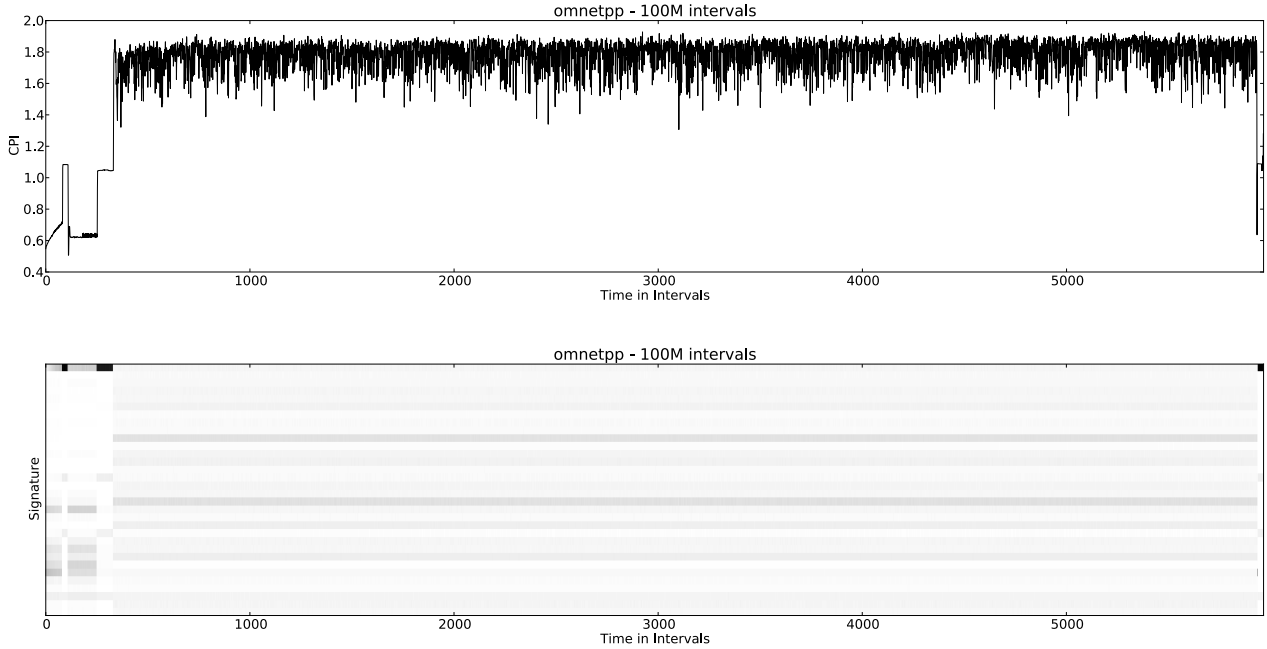


Figure B.42: CPI and the Signature for *omnetpp*.

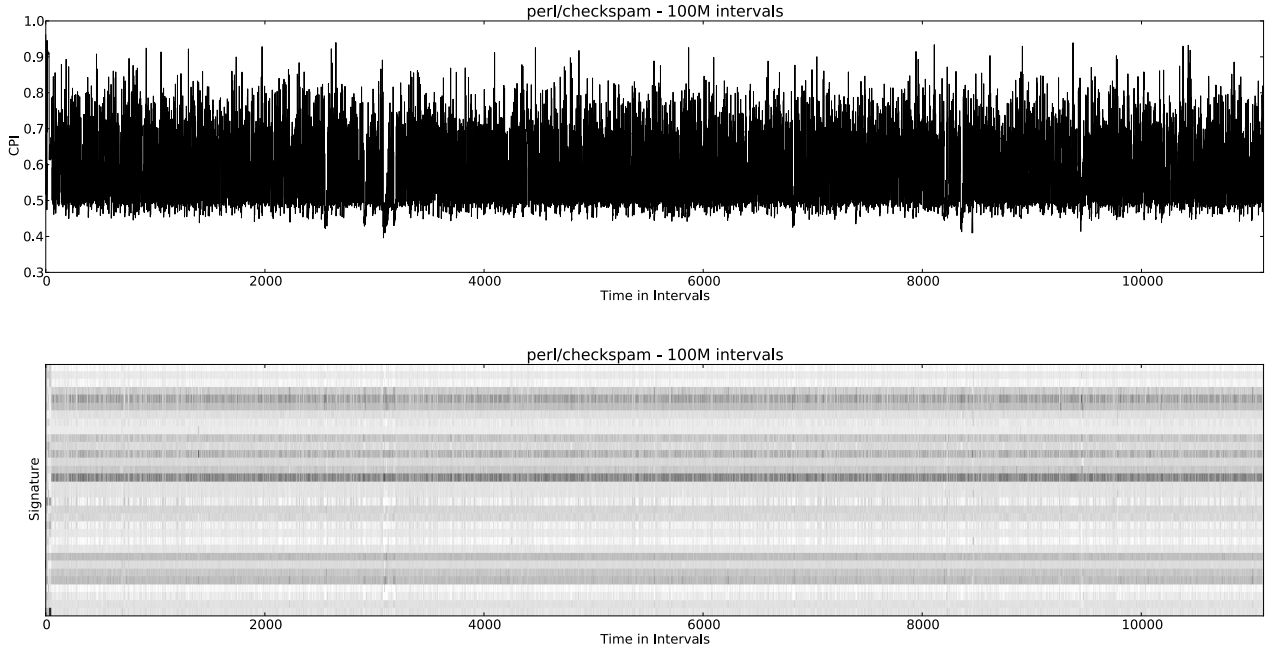


Figure B.43: CPI and the Signature for *perl/checkspam*.

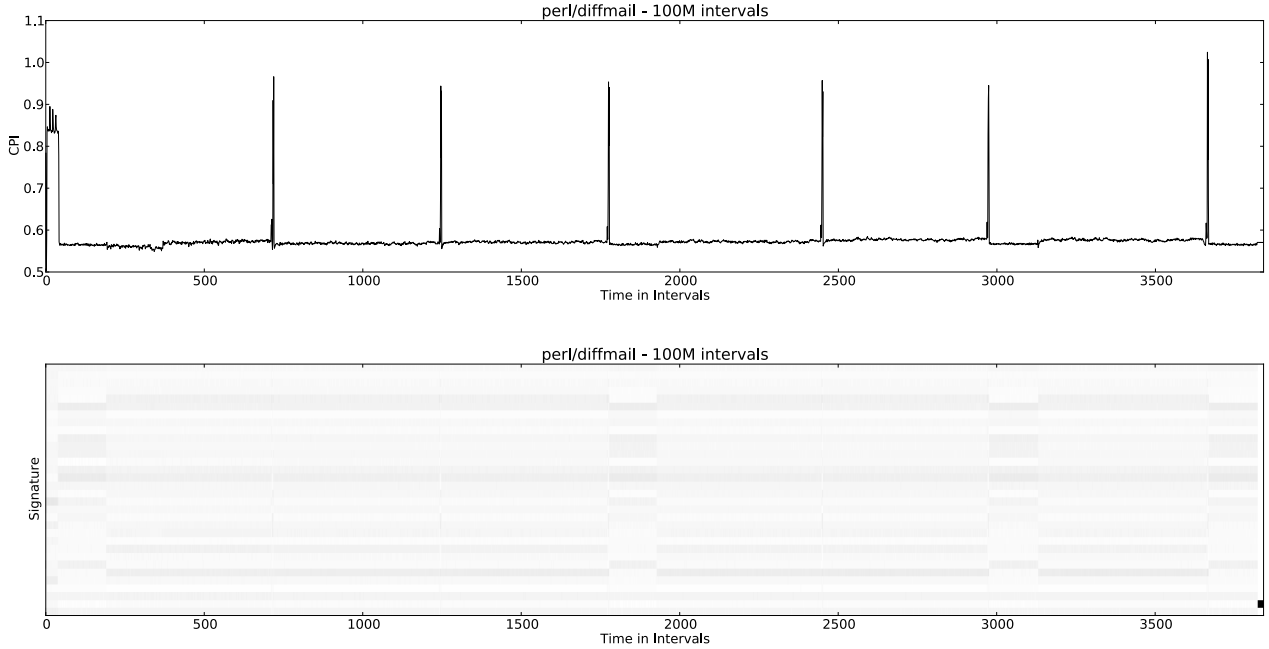


Figure B.44: CPI and the Signature for *perl/diffmail*.

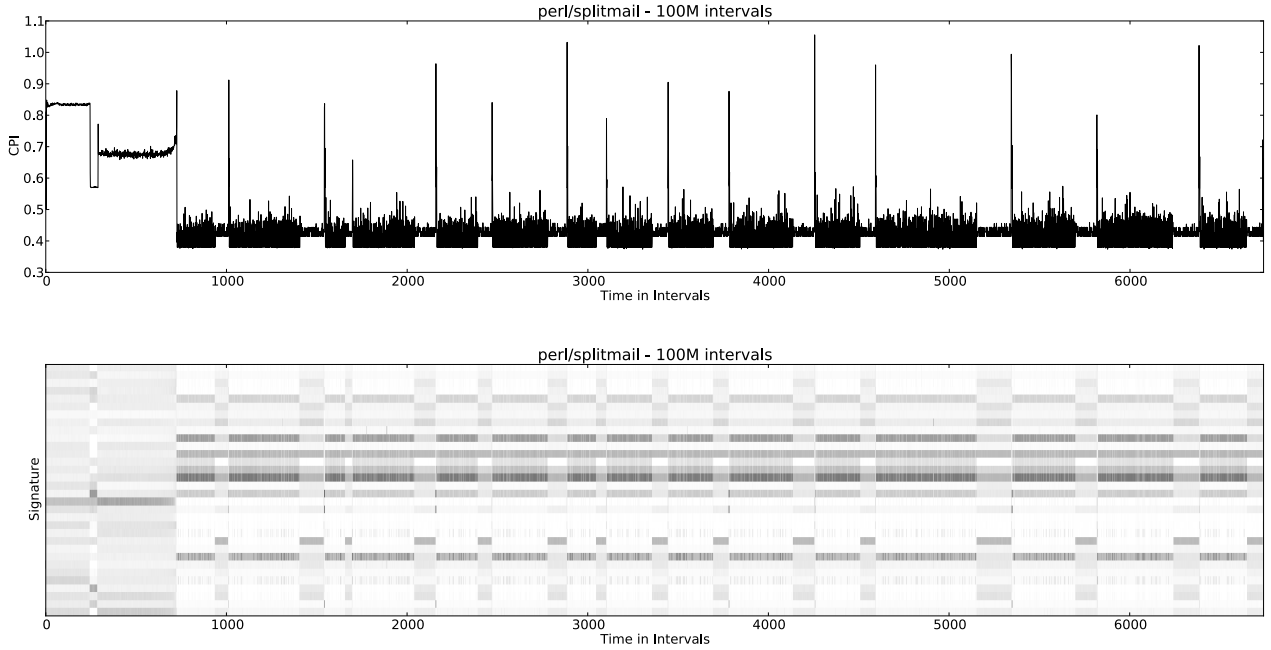


Figure B.45: CPI and the Signature for *perl/splitmail*.

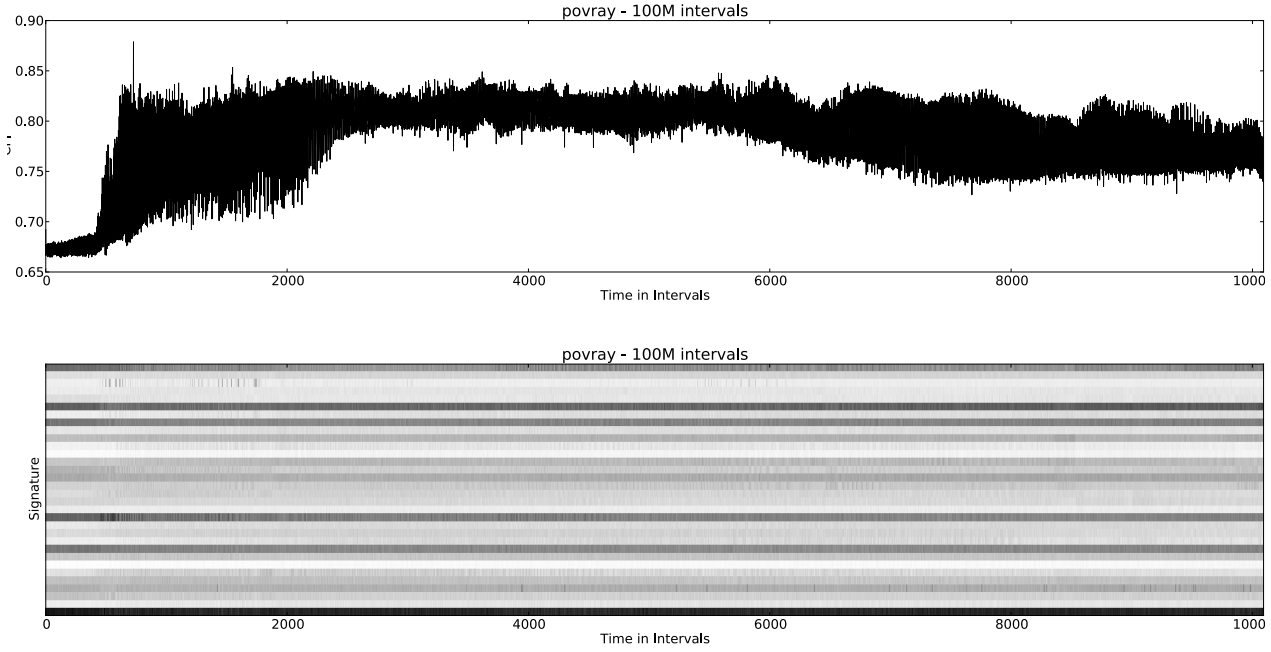


Figure B.46: CPI and the Signature for *povray*.

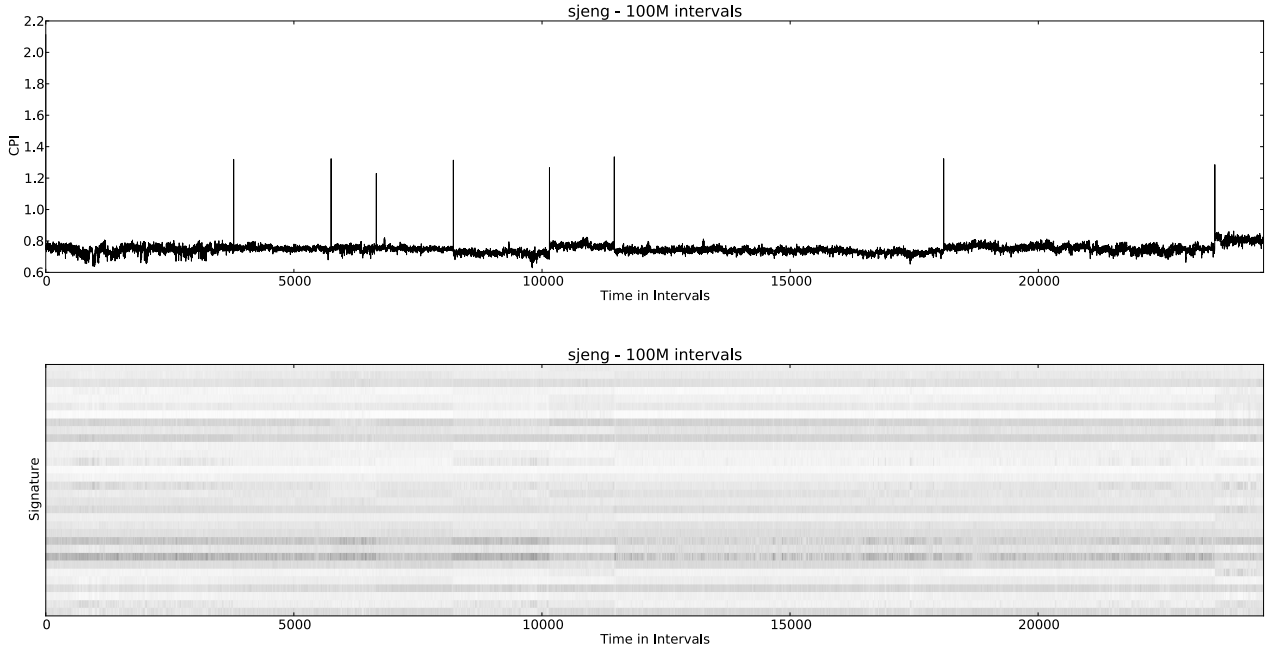


Figure B.47: CPI and the Signature for *sjeng*.

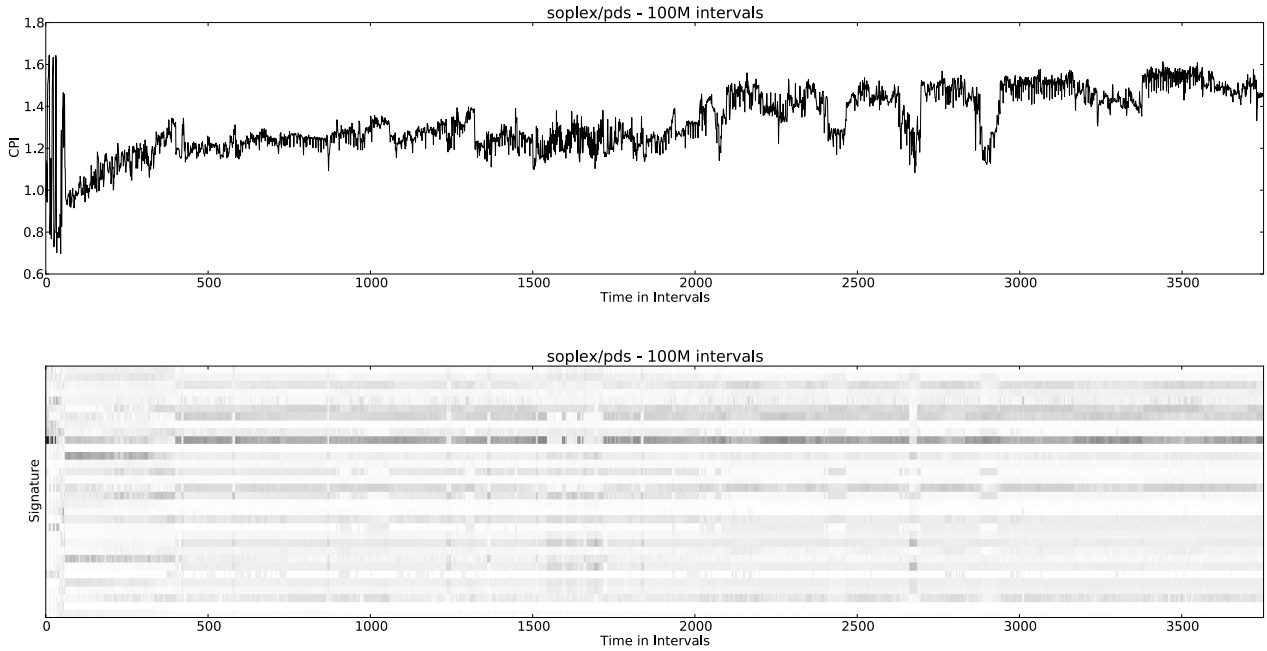


Figure B.48: CPI and the Signature for *soplex/pds*.

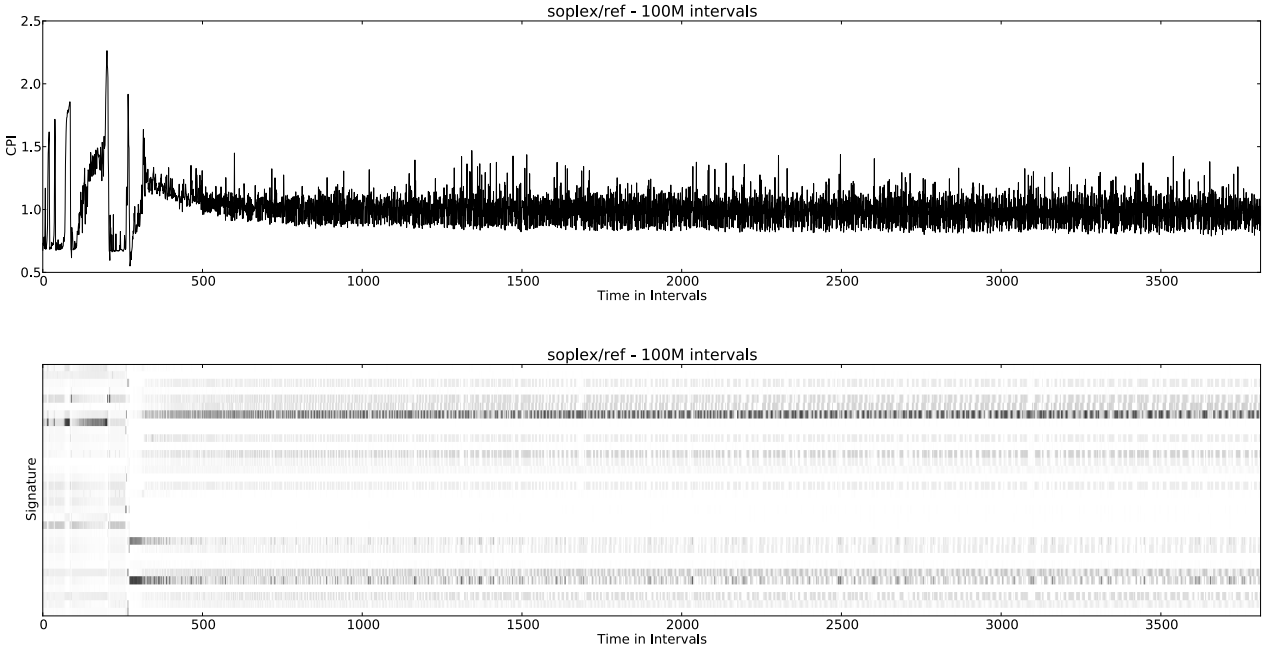


Figure B.49: CPI and the Signature for *soplex/ref*.

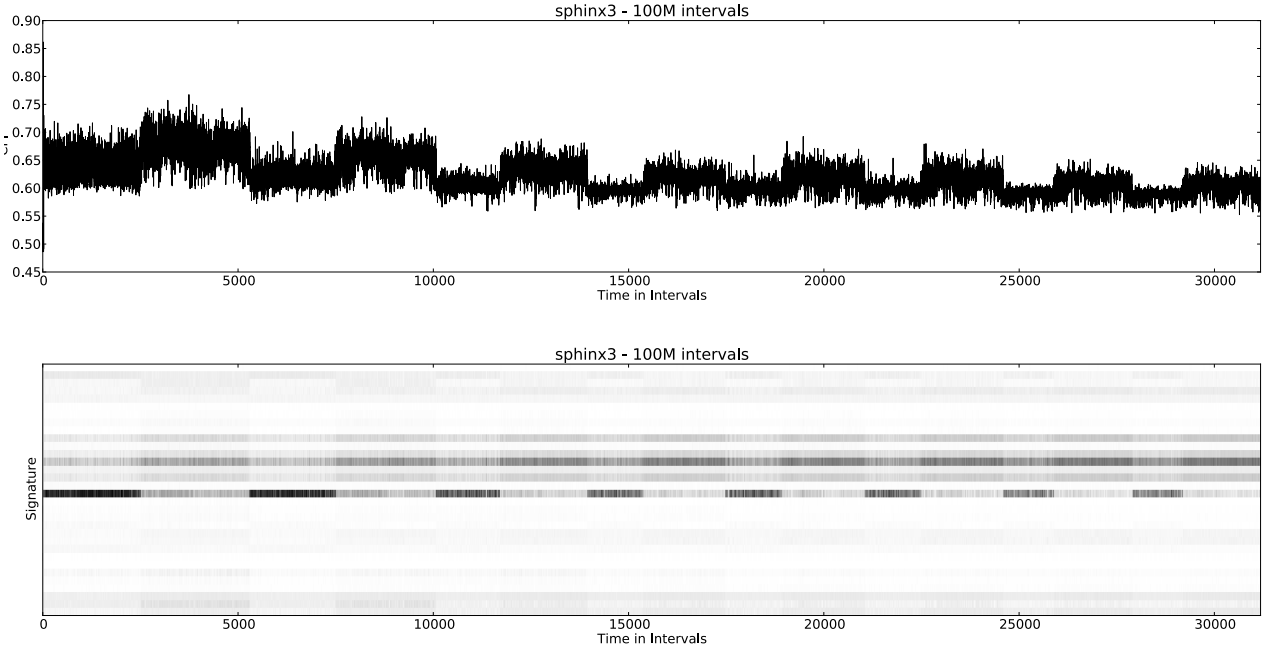


Figure B.50: CPI and the Signature for *sphinx3*.

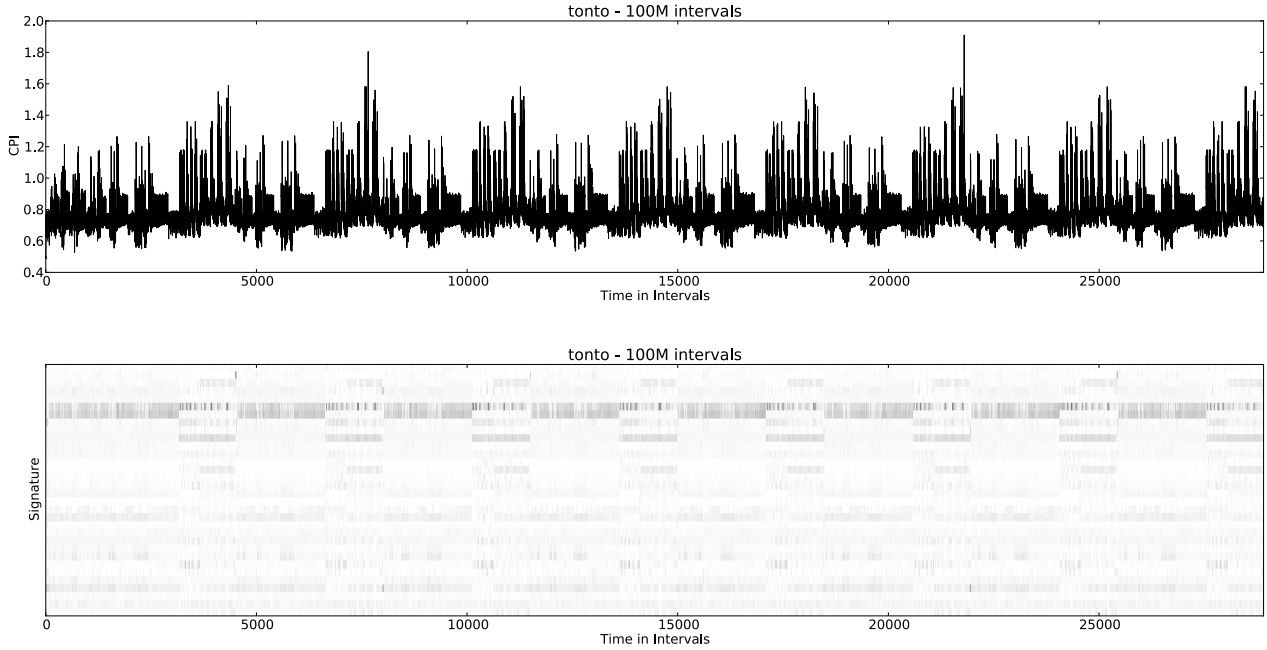


Figure B.51: CPI and the Signature for *tonto*.

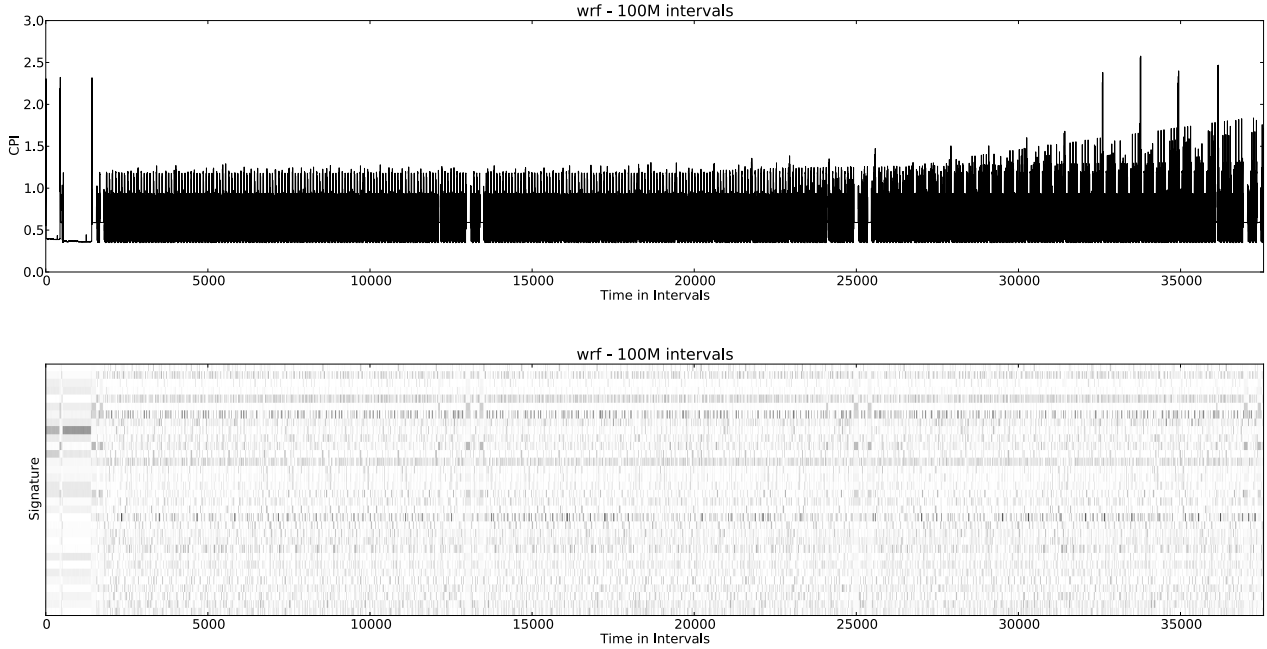


Figure B.52: CPI and the Signature for *wrf*.

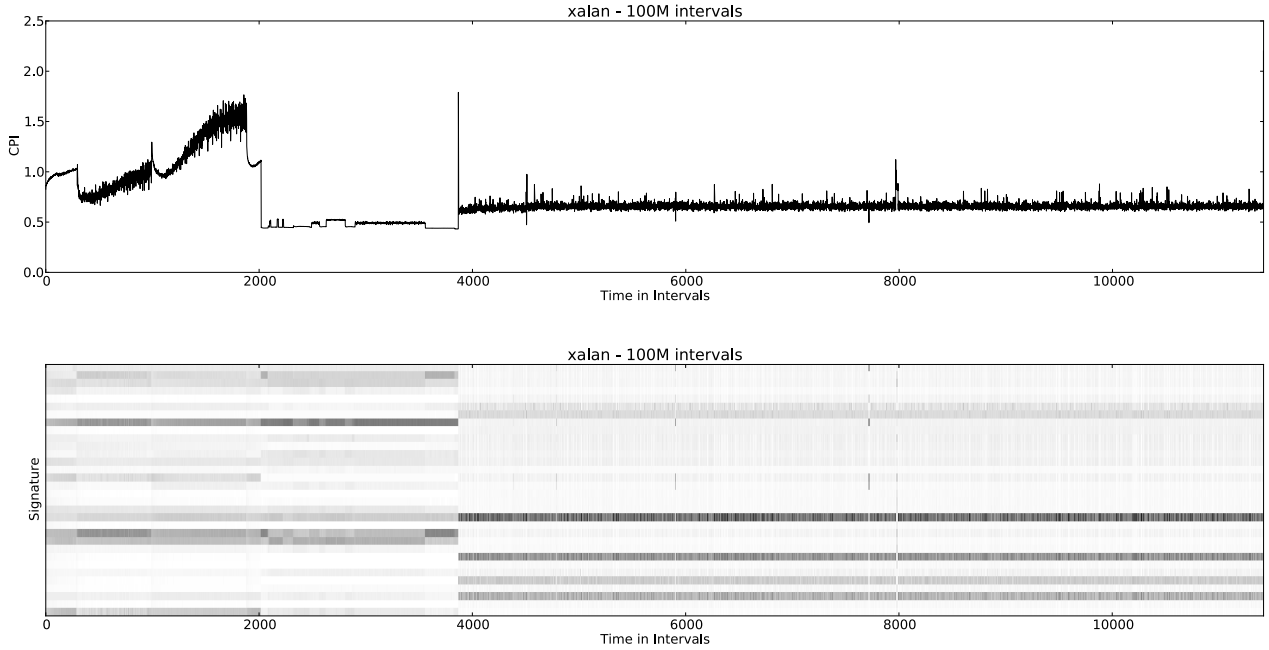


Figure B.53: CPI and the Signature for *xalan*.

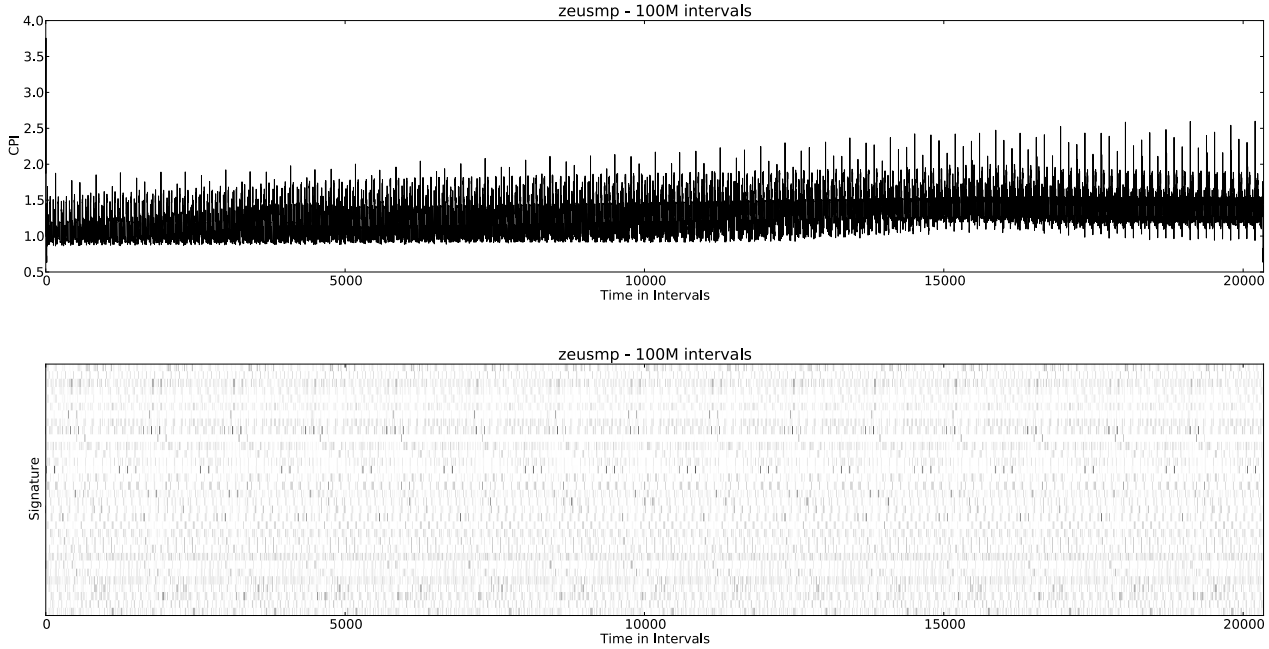


Figure B.54: CPI and the Signature for *zeusmp*.

Listings

looppac.h

```
1  /**
2   * @file
3   * @author Andreas Sembrant<andreas@sembrant.com>
4   *
5   */
6
7  #ifndef __LOOPPAC_H
8  #define __LOOPPAC_H
9
10 #ifdef __cplusplus
11 extern "C" {
12 #endif
13
14 #include <stddef.h>
15 #include <stdint.h>
16 #include <signal.h>
17
18 /**
19  * @brief Event information sent with the callbacks.
20  */
21 struct looppac_event_info
22 {
23
24     /**
25      * @brief A handle to the monitor.
26      */
27     void *handle;
28
29     /**
30      * @brief The thread id.
31      */
32     int tid;
33
34     /**
35      * @brief The phase id.
36      */
37     int phase;
38
```

```
39      /**
40       * @brief The etimated time to the next phase change.
41       */
42      int etpc;
43
44      /**
45       * @brief Prediction of the coming interval.
46       */
47      struct prediction
48      {
49
50          /**
51           * @brief The phase id of the next interval.
52           */
53          int phase;
54
55          /**
56           * @brief How confident we are in our prediction.
57           */
58          size_t confidence;
59
60      } prediction;
61
62      /**
63       * @brief The size of the interval as a multiple of the base length.
64       *
65       * With dynamic interval the size of the interval may grow. This
66       * variable tells how much the interval has been grown with.
67       */
68      size_t interval_size;
69
70      /**
71       * @brief The sample that was taken during the interval.
72       */
73      struct sample
74      {
75
76          /**
77           * @brief A sample in the interval.
78           */
79          uint64_t sample;
80
81          /**
82           * @brief The number of instructions in the insterval.
83           */
84          uint64_t instructions;
85
86      } sample;
87
88      /**
89       * @brief The signature of the interval.
90       */
91      struct signature
```

```
 92|     {
 93|
 94|         /**
 95|          * @brief The size of the vector.
 96|          */
 97|         size_t size;
 98|
 99|         /**
100|          * @brief The signature vector.
101|          */
102|         const double *vector;
103|
104|     } signature;
105|
106| };
107|
108| /**
109|  * Typedef for the info struct.
110|  */
111| typedef struct looppac_event_info looppac_event_info_t;
112|
113| /**
114|  * @brief Called when an interval is over.
115|  *
116|  *                   .------- phase
117|  *                  |  .----- time when the callback will be called
118|  *                  | |  .--- prediction
119|  *                  ! ! !
120|  * ... 2 | 1 | 1 | 2 | ? | ? | ? ...
121|  *
122|  * @param info->tid[in]           The thread id.
123|  * @param info->phase[in]         The phase the interval was in.
124|  * @param info->prediction[in]    The predicted phase id of the next
125|  *                                interval.
126|  * @param v                       User defined data.
127|  *
128|  * @reentrant The function must be reentrent.
129|  */
130| typedef void (*interval_callback_t)(const looppac_event_info_t *info,
131|                                     void *v);
132|
133| /**
134|  * @brief Called when a change has occured.
135|  *
136|  *             .----- time when the callback will be called
137|  *            |
138|  *            !
139|  * ... 2 | 1 | ? | ? | ? ...
140|  *
141|  * @param info->tid[in]           The thread id.
142|  * @param info->phase[in]         The phase the interval was in.
143|  * @param info->prediction[in]    The predicted phase id of the next
144|  *                                interval.
```

```
145  * @param v                          User defined data.
146  *
147  * @reentrant The function must be reentrent.
148  */
149 typedef void (*phase_changed_callback_t)(const looppac_event_info_t *info,
150                                          void *v);
151
152 /**
153  * @brief Monitor settings.
154  */
155 struct looppac_monitor_attr
156 {
157     /**
158      * @brief The sample period is the number of branches between samples.
159      *        A low sample_period results in a haig sample rate.
160      */
161     uint64_t sample_period;
162
163     /**
164      * @brief The size of the execution interval in number of instructions.
165      */
166     uint64_t interval_size;
167
168     /**
169      * @brief The user context that is sent along with the callbacks.
170      * @see interval_callback_t and phase_changed_callback_t
171      */
172     void *user_context;
173
174     /**
175      * @brief Called when an interval is over.
176      */
177     interval_callback_t interval_callback;
178
179     /**
180      * @brief Called when a change has occured.
181      */
182     phase_changed_callback_t phase_changed_callback;
183
184     /**
185      * @brief Linux perf_events settings.
186      */
187     struct perfevent
188     {
189
190         /**
191          * @brief The branch counter id.
192          */
193         int branch_counter;
194
195         /**
196          * @brief The branch counter type id.
197          * @see enum perf_type_id in perf_events.h;
```

```
198             */
199            int branch_type_id;
200
201            /**
202             * @brief Randomize the sample period.
203             * @warning The kernel must have been patch for this to work.
204             */
205            int randomize;
206
207            /**
208             * @brief Enable/disable PEBS.
209             * @warning The kernel must have been patch for this to work with
210             *          branch instructions.
211             */
212            int precise_ip;
213
214            /**
215             * @brief Extra perfomance counter that can be used to sample
216             *        hardware statistics during the execution interval.
217             */
218            int ext_sample_counter;
219
220            /**
221             * @brief Extra perfomance counter type id.
222             * @see enum perf_type_id in perf_events.h;
223             */
224            int ext_sample_type_id;
225
226            /**
227             * @brief Enable the extra sample counter.
228             */
229            int enable_ext_sample;
230
231        } perfevent;
232
233        /**
234         * @brief Interval policy.
235         */
236        struct interval
237        {
238
239            #define LOOPPAC_INTERVAL_TYPE_FIXED      0
240            #define LOOPPAC_INTERVAL_TYPE_DYNAMIC    1
241
242            /**
243             * @brief The type of interval;
244             */
245            int type;
246
247            union
248            {
249                struct fixed
250                {
```

```
251
252             } fixed;
253
254             struct dynamic
255             {
256
257                 /**
258                  * @brief The scaling factor.
259                  */
260                 int multiplier;
261
262                 /**
263                  * @brief The maximum size of the interval.
264                  */
265                 int maximum_size;
266
267             } dynamic;
268
269         } attr;
270
271     } interval;
272
273     /**
274      * @brief Classifier.
275      */
276     struct classifier
277     {
278
279         #define LOOPPAC_CLASSIFIER_TYPE_DISTANCE_BASED      0
280         #define LOOPPAC_CLASSIFIER_TYPE_SEQUENTIAL_KMEANS   1
281
282         /**
283          * @brief The type of classifier;
284          */
285         int type;
286
287         union
288         {
289
290             /**
291              * @brief Distance based classification
292              */
293             struct distance_based
294             {
295
296                 /**
297                  * @brief The similarity threshold, [0,100]
298                  */
299                 double similarity_threshold;
300
301                 /**
302                  * @brief Number of intervals a phase must have been seen in
303                  *        before beeing assigned a unique phase id.
```

```
304                  */
305                 int transition_threshold;
306 
307             } distance_based;
308 
309             /**
310              * @brief Sequential K-Means Classification
311              */
312             struct seq_kmeans
313             {
314 
315                 /**
316                  * @brief The similarity threshold, [0,100]
317                  * @note Distance based classification is used until we have
318                  *       K clusters.
319                  */
320                 double similarity_threshold;
321 
322                 /**
323                  * @brief The number of clausters.
324                  */
325                 int k;
326 
327                 /**
328                  * @brief The learning rate.
329                  */
330                 int learning_rate;
331 
332             } seq_kmeans;
333 
334         } attr;
335 
336     } classifier;
337 
338     /**
339      * @brief Predictor.
340      */
341     struct predictor
342     {
343 
344         #define LOOPPAC_PREDICTOR_TYPE_LAST_VALUE      0
345         #define LOOPPAC_PREDICTOR_TYPE_MARKOV          1
346         #define LOOPPAC_PREDICTOR_TYPE_PPM             2
347         #define LOOPPAC_PREDICTOR_TYPE_RUN_LENGTH      3
348 
349         /**
350          * @brief The type of predictor;
351          */
352         int type;
353 
354         union
355         {
356 
```

```
357             /**
358              * @brief Last value predictor.
359              */
360             struct last_value
361             {
362
363             } last_value;
364
365             /**
366              * @brief Markov predictor
367              */
368             struct markov
369             {
370
371                 /**
372                  * @brief The size of the phase history.
373                  */
374                 int phase_history_size;
375
376                 /**
377                  * @brief The number of correct prediction before relying on
378                  *        the prediction. Otherwise the last valu prediction
379                  *        is used.
380                  */
381                 int confidence_threshold;
382
383                 /**
384                  * @brief The size of the cache
385                  */
386                 int cache_size;
387
388             } markov;
389
390             /**
391              * @brief Run length predictor
392              */
393             struct run_length
394             {
395
396                 /**
397                  * @brief The number of correct prediction before relying on
398                  *        the prediction. Otherwise the last valu prediction
399                  *        is used.
400                  */
401                 int confidence_threshold;
402
403                 /**
404                  * @brief The size of the cache
405                  */
406                 int cache_size;
407
408             } run_length;
409
```

```
410         } attr;
411
412     } predictor;
413 };
414
415 /**
416  * Typedef for the monitor attr struct.
417  */
418 typedef struct looppac_monitor_attr looppac_monitor_attr_t;
419
420 /**
421  * @brief Init the library.
422  * @param The log file.
423  * @returns 0 on success, otherwise -1.
424  */
425 extern int looppac_init(const char *log);
426
427 /**
428  * @brief Shutdown the library and free any allocated resources.
429  *
430  * @returns 0 on success, otherwise -1.
431  */
432 extern int looppac_shutdown();
433
434 /**
435  * @brief A string representation of the error.
436  * @param monitor[in]   Monitor handle. If null get system error.
437  */
438 extern const char *looppac_strerror(void *monitor);
439
440 /**
441  * @brief Create a monitor for a given thread.
442  * @param tid[in]   The thread/process id.
443  * @param attr[in]  Monitor settings.
444  * @returns NULL on failure, otherwise a monitor handle.
445  */
446 extern void *looppac_open_monitor(int tid, looppac_monitor_attr_t *attr);
447
448 /**
449  * @brief Close a monitor.
450  * @param monitor[in]   Monitor handle.
451  * @returns 0 on success, otherwise -1.
452  */
453 extern int looppac_close_monitor(void *monitor);
454
455 /**
456  * @brief Start monitoring the phases.
457  * @param monitor[in]   Monitor handle.
458  * @returns 0 on success, otherwise -1.
459  */
460 extern int looppac_start_monitor(void *monitor);
461
462 /**
```

```
463  * @brief Stop monitoring the phases.
464  * @param monitor[in]   Monitor handle.
465  * @returns 0 on success, otherwise -1.
466  */
467 extern int looppac_stop_monitor(void *monitor);
468
469 /**
470  * @brief Forward a signal.
471  * @param signum[in]   The signal number.
472  * @param info[in]     The info block.
473  * @param uc[in]       User context.
474  *
475  * @returns 0 if the signal was caught, 1 if the signal did not belong to
476  *          looppac, -1 on error.
477  *
478  * The user of this library should have registered a signal handler for SIGIO.
479  * For each interval a signal will be generated. Pass on that signal to the
480  * looppac.
481  *
482  * @warning All signals must be forwarded. If a signal destined to looppac is
483  *          ignored, unspecified behaviour will occur.
484  *
485  */
486 extern int looppac_forward_signal(int signum, siginfo_t *info, void *uc);
487
488 #ifdef __cplusplus
489 }
490 #endif
491
492
493 #endif /* __LOOPPAC_H */
```