Technical Report: Feedback-Based Generation of Hardware Characteristics

Marcus Jägemar
Sigrid Eldh
Andreas Ermedahl
Ericsson AB
first.last at ericsson.com

Björn Lisper
Mälardalen University
bjorn.lisper at mdh.se

ABSTRACT

In large complex server-like computer systems it is difficult to characterise hardware usage in early stages of system development. Many times the applications running on the platform are not ready at the time of platform deployment leading to postponed metrics measurement. In our study we seek answers to the questions: (1) Can we use a feedback-based control system to create a characteristics model of a real production system? (2) Can such a model be sufficiently accurate to detect characteristics changes instead of executing the production application?

The model we have created runs a signalling application, similar to the production application, together with a PID-regulator generating L1 and L2 cache misses to the same extent as the production system. Our measurements indicate that we have managed to mimic a similar environment regarding cache characteristics. Additionally we have applied the model on a software update for a production system and detected characteristics changes using the model. This has later been verified on the complete production system, which in this study is a large scale telecommunication system with a substantial market share.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; B.3.2 [**Memory Structures**]: Design styles—*Cache Memories*; C.4 [**Performance of Systems**]: Measurement techniques

General Terms

Measurement, Performance

Keywords

Control Theory, Feedback computing, Performance Analysis, Characteristics, Cache memories, Simulation, Load Testing and Design Aids

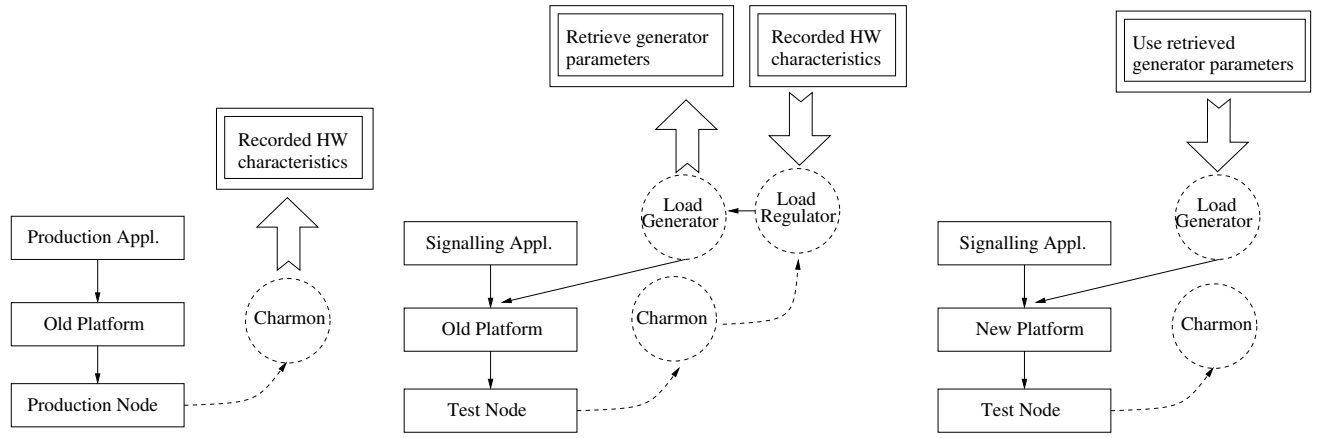
1. INTRODUCTION

Measuring behavioural characteristics for complex large scale computer systems is difficult since this requires either a full production system or advanced test programs with large test systems. After a software update, it is essential to measure behavioural characteristics and check that the nature of the system has not changed. Behavioural changes results in costly and time consuming verification in the development cycle. Late detection of unfulfilled requirements due to characteristics changes leads to increased lead time, since parts of the system must be re-investigated and re-implemented. Such increase in development time may not be accepted since short time-to-market is essential [16, 7, 13, 19, 20].

The system we are investigating is a telecommunication system with a market share of about 38% in 2011 [7]. It consists of 5M SLOC [4] and runs on more than 20 types of boards with different hardware layout and functionality servicing both voice and data communication.

We define one instance of the computer system we are investigating as a *node*. One node can consist of many CPU's but from a system point of view they are grouped as one execution entity. A large scale node has many CPU's, a small scale node may consist of only a single CPU. A node communicates extensively both internally and externally between nodes using signals, i.e., operating system messages. We introduce two concepts central to our investigation. The first; *behavioural characteristics* is in our case CPI, CPU-load or signal turnaround time but can be any metric that describes the behaviour or performance of the system. The second is *load characteristics* which is described by metrics that will affect the behaviour characteristics of the system. In our investigation we have concentrated on cache misses but it can be any other metric such as TLB usage, branch statistics, number of system calls or interrupts etc.

For early detection of behavioural characteristics changes we suggest to create a model of the production system on a small scale node. The benefit of doing so is that we don't have to wait for the availability of large scale nodes which are expensive and difficult to obtain. Additionally, changes in the platform may require modifications in the application software which even more extends the time before characteristics measurements can be made. Our approach is to alter load characteristics, in our case cache miss rate, to change the behavioural characteristics. Our model system consists



a) Recording characteristics from a production system

b) Creating a simulation environment from previously recorded characteristics and retrieve regulator settings

c) Observe characteristics for a new platform by using signalling application and load generator

Figure 1: Our procedure in three steps to measure characteristics for a new software release by modelling a production system.

of a signalling application and a load regulator. The signalling application simulates the production system by communicating extensively between processes. Additionally, it is used to detect performance differences when applying the model, i.e. instead of using IPC/CPI as a metric. The load regulator is implemented as a Proportional-Integral-Derivative (PID) regulator and generates hardware load in the form of cache misses at a rate consistent with a real production system. Changes in the production system can now be tested on the model system with similar behaviour. The model in itself is easily extendable with additional regulators and generators for additional hardware metrics such as L3-cache hit/miss rate, branch prediction statistics etc. The results we have achieved is to PID-regulate L1 Instruction, L1 Data and L2 Data cache misses according to a predefined rate measured on a real production system. Additionally we have succeeded to detect behavioural characteristics changes in the production environment by using the model node to test a new software release.

The rest of the paper is organised as follows: In Section 2 we present our process using a regulator to mimic behavioural characteristics on a model node. We have described the details of our work in Section 3 together with the regulation algorithm and load generation methods. Section 4 reveals our experimental results when modelling an environment and a real-life test measuring performance changes between two releases of a system. In Section 5 we give some directions to related work. Section 6 suggests future work and possible improvements as well as shortcomings we have detected.

2. PROCESS

We define the *platform* as the operating system bundled with basic cluster functionality such as program handling, error recovery mechanism, load balancing etc. On top of the platform one or several telecommunication applications run using the platform API. The *system* denotes the complete executing software on the hardware. In our investigation we have followed this procedure to model a large system on a much smaller node.

1. Record characteristics for an existing production system, see Figure 1a.
   (a) Run a complete customer system for which we want to create a hardware (HW characteristics model.
   (b) Measure load-characteristics, in our case L1 Data, L1 Instruction and L2 Data cache misses.
2. Create a simulation environment that mimics the characteristics, obtained in step 1, on a test node, see Figure 1b.
   (a) Start the platform (same release as in step 1) together with the signalling application.
   (b) Use the HW load regulation algorithm to reach the same load-characteristics ratio as in step 1b.
   (c) Retrieve metrics from the regulation algorithm. In our case the internal counters used to describe the amount of cache-misses generated by the load generation algorithm.

In the scenario above we have sampled behavioural characteristics from a real customer system and then recreated a similar execution environment for a limited platform signalling application. By storing the internal load-generation parameters, in 2c, we can generate the same rate of cache misses without using the regulation algorithm. This allows us to change the platform and then apply the same rate of misses. Investigating the ratio of misses allows us to detect changes in platform behaviour. In the continued procedure below we can measure characteristics for a different release of the platform to get an indication of how it will perform running the complete system.

3. Detecting behavioural characteristics changes, such as signal turn-around time, on small scale hardware, see Figure 1c.
   (a) Start the new platform together with the signalling application.
   (b) Generate HW-load at the same rate as obtained in step 2c.

(c) Check behavioural characteristics for the benchmarking application, such as signal turnaround time.

3. OUR APPROACH

3.1 Hardware and Software Resources

The operating system used in the platform investigated by us is a small embedded real-time OS running on a Freescale processor. The cache related details according to Freescale reference manuals [11, 12] are:

- L1 Cache Size : 32kB separate 8-way set associative I and D cache per core with 64B cache line size.
- L2 Cache Size : 128kB common 8-way set associative I and D cache per core.
- L3 Cache Size 2MB common I and D cache for all cores.

All cache levels uses pseudo LRU replacement strategy.

3.2 The Characteristics Monitor

Implemented within the platform is a continuously running characteristics monitor called charmon. It gathers information about the HW-usage of the complete system by periodically sampling performance monitor counters, PMS's. PMC's can be configured to count different HW-events such as cache misses, TLB misses, branch statistics etc. The charmon stores counted events in a database together with commonly requested key performance indices such as cycles per instruction (CPI), L1-2-3 cache hit/miss rate and ratio, TLB hit/miss rate and ratio, branch statistics, CPU load and others. The probe effect is low since the PMC's are located inside the CPU with low or no performance penalty and the database storage and PMC reprogramming occurs infrequently.

In our platform we have currently implemented 13 different sets of performance counters. The procedure when measuring characteristics is as follows; first a set of PMC counters is programmed into the PMC and triggered to start counting. Then the character monitor sleeps for 1 second until woken by a timer. Then the counters are read and stored in the database and another set of performance counters are programmed into the PMC and started. This gives each counter set a periodicity of 13 seconds and length of 1 second. The periodicity and sample length has its limitations but shortening the time to reduce the granularity increases the intrusiveness, which negatively affects the data collected. In our system the sample length of 1 second is sufficiently good for the relatively stable load generated.

Furthermore, using PMS's gives us the opportunity to measure non-instrumented code reducing the intrusiveness of the monitor. Other CPU architectures such as x86 implements performance counting in a performance monitor unit described by Eranian [21], which could be used for the same purposes as the PMC. For a general description of the PMC's see the Freescale P4080 reference manual [11] and for more details the e500mc core reference manual [12], specifically Table 9-47.

3.3 The Load Regulator

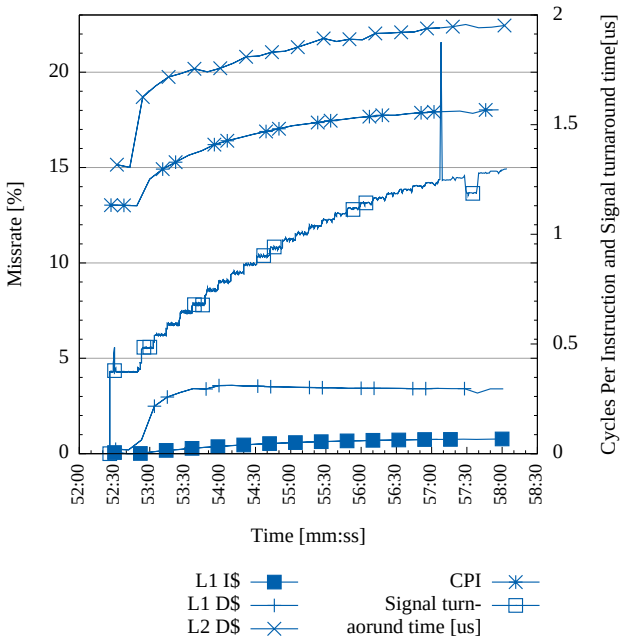


Figure 2: Cache misses and CPI when running a simple signalling application bouncing signals between two processes located on the same core. The load regulation application strives to have a system miss rate of 0.74% L1 I-cache, 3.3% L1 D-cache and 22% L2 D-cache which is similar to a production system. The regulation scenario described in this figure was started after the system was completely stable, in our case after about 52:00 minutes

The load regulator operates in two modes. The first mode is a client to charmon subscribing to metrics for certain HW-properties. With a user supplied ratio as reference the regulator tries to generate HW-load reaching the reference value. In our investigation we have implemented regulation algorithms for L1 Instruction, L1 Data and L2 Data cache misses. The second mode operates in a stand-alone manner generating cache misses at a specific rate without any feedback of current metrics.

3.4 The Control Algorithm

To generate a specific HW-load ratio a PID-regulator is used to control each parameter. Each HW-property, such as cache miss-rate, is controlled by its own PID-regulator. See Figure 2 for a typical regulation scenario. The test application has an initial characteristics (to the left) that differs from the final characteristics reached after the regulation has started converging. The effect of increasing number of cache misses can be observed by looking at the CPI and signal turn-around time that increases as the cache usage is increased. The spike (57:00) in the graph is difficult to explain but there are many services running on the system being monitored. It can be any periodically running signalling service synchronising information with other boards generating a burst of load causing a spike. However, the regulation algorithm keeps converging after the spike has occurred. The initial control loop parameters for each of the properties were empirically discovered to provide stability rather than quick convergence.

Listing 1: The function that will generate the actual ICache misses.

```
1 int bigswitch(int n) {
2   switch (n) {
3     case 1: n += 10;
4       break;
5     case 2: n += 11;
6       break;
7     case 3: n += 11;
8       break;
9     ...
10    case 99999: n += 50009;
11      break;
12    default: n+= 20;
13  }
14   return n;
15 }
```

Listing 2: The disassembly (objdump -D) of .text section for PowerPC generated code with gcc 4.4.3.

```
1       ...
2       9c: 48 24 9e b0   b       249f4c <
            bigswitch+0x249f4c>
3       a0: 80 1f 00 08   lwz     r0,8(r31)
4       a4: 3d 60 00 02   lis     r11,2
5       a8: 7c 00 5a 14   add     r0,r0,r11
6       ac: 30 00 86 a2   addic   r0,r0
            ,-31070
7       b0: 90 1f 00 08   stw     r0,8(r31)
8       b4: 48 24 9e 98   b       249f4c <
            bigswitch+0x249f4c>
9       b8: 80 1f 00 08   lwz     r0,8(r31)
10      bc: 3d 20 00 02   lis     r9,2
11      c0: 7c 00 4a 14   add     r0,r0,r9
12      c4: 30 00 86 a2   addic   r0,r0
            ,-31070
13      c8: 90 1f 00 08   stw     r0,8(r31)
14      cc: 48 24 9e 80   b       249f4c <
            bigswitch+0x249f4c>
15      ...
```

3.5 Generating L1 I-cache Misses

To generate instruction cache misses, a method using a large switch-case statement is utilised. A function is called with varying argument values for the switch-case index thus generating an I-cache hit if the distance, i.e., the number of executable instructions in the code flow, from the previous call is short and a miss if the distance is longer. As can be seen in Listing 2 each case statement in Listing 1 generates 24B of instructions. This way to generate I-cache misses was suggested in [18].

3.6 Generating L1 and L2 D-cache misses

The L1 Data and L2 Data cache misses are generated by using different strides through memory. Depending on the desired rate of L1 Data and L2 Data misses the tags and sets in the cache memory will be excercised differently. In the Freescale address translationa Physical Address (PA) is represented by 36bits. For each cache set eight address tags are stored with the corresponding 64B cache line. The selection of address tag is done with bits PA[0:23] and set selection with PA[24:29]. To select a specific word inside the cache line PA[30:35] is used, see [12].

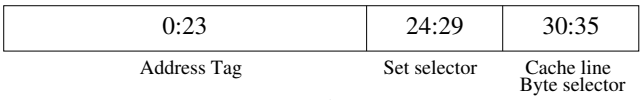


Figure 3: PowerPC L1 Cache Address structure for sets and tags.

Listing 3: The function that generates L1 D-cache and L2 D-cache misses.

```
1 #define TAGS (8)
2 #define SETS (64)
3 for(di=iter; di>0; di--) {
4  unsigned int k;
5  unsigned int set;
6  unsigned int tag;
7  for(k=0; k<TAGS; k++) {
8   if((*areaToggleCnt % (unsigned int)
        offsRatio)==0) {
9      if(*offsetCnt >= 7)
10      *offsetCnt = 0;
11     else
12      (*offsetCnt)++;
13     *offset = (64*1024) * *offsetCnt;
14  }
15  if(*areaToggleCnt > MAX_AREA_TOGGLE_CNT)
16   *areaToggleCnt=0;
17  else
18   (*areaToggleCnt)++;
19  for(i=setstart; i<SETS; i++) {
20   set = (i << 7);
21   tag = (64*k) << 13;
22   pointer = ptr[core];
23   if(pointer != NULL) {
24    a += pointer[(*offset + set + tag)/4];
25   }}}}
```

The algorithm described in Listing 3 shows one way to generate L1 and L2 D-cache misses by first iterating over all address tags in a cache set (all 8 of them) and then iterate over all sets in the cache (all 64). To be able to regulate the number of L1 D-cache misses there are two variables that are modified by the regulation algorithm. The most important variable is *di*, which determines the number of iterations. For smaller adjustments when a small cache miss rate is requested the iteration count will be set to a small value and the *setstart* variable will be used to select the number of sets to walk through. Increasing this variable will reduce the number of misses further since fewer sets will be evicted by this loop. To generate L2 D-cache misses the area toggle is introduced. The idea is that the basic algorithm will work more or less within the scope of the L1 D-cache space. Adding a offset causing the memory address to access to be outside that of L1 at a certain ratio will increase the L2 D-cache miss rate. One problematic issue is that introducing L2 cache misses affects the number of L1 misses so the regulation algorithm will then need to recalculate the other regulation parameters.

4. RESULTS

4.1 Mimicking a Production Node Environment

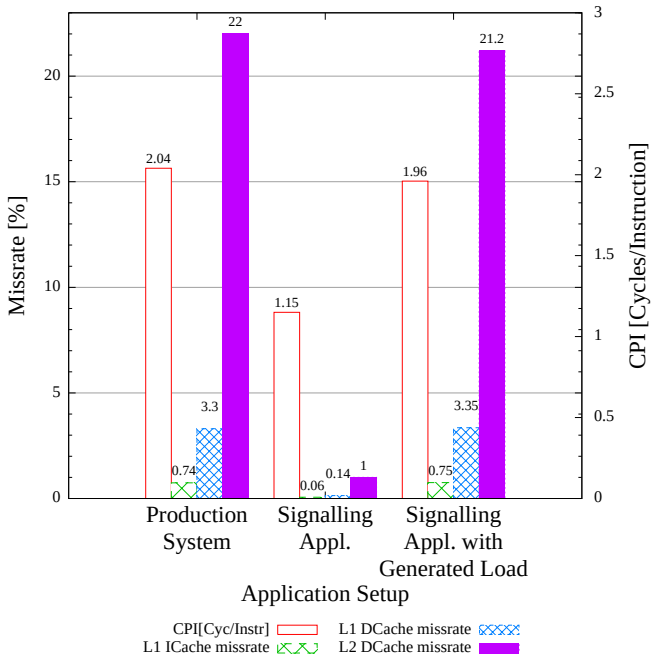


Figure 4: Measured cache usage and CPI for the reference system, test application and a recreated execution environment with the test application. Characteristics for the reference system is similar to the mimicked scenario.

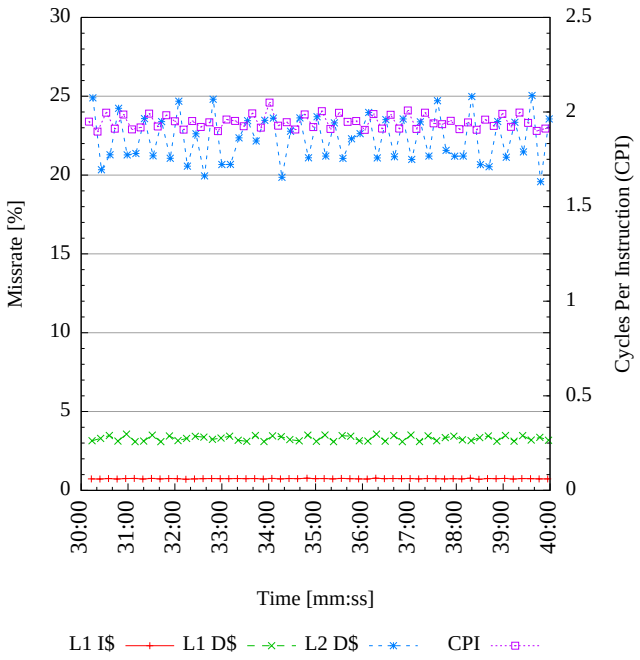


Figure 5: Cache misses and CPI when running a test application and the load generator trying to recreate a traffic scenario similar to a real live node.

The assumption is that introducing cache misses will cause the application to execute in a less efficient way. According to Doucette and Federova [5] an introduction of L2 D-cache load causes a significant slowdown of simultaneously running applications. We have used cache misses to mimic the execution environment of a real customer based system within the constraints of a much smaller test suite. A first test of the procedure outlined in Section 2 reveals that a load regulator can add a hardware load yielding a similar load profile as the real application. As can be seen in Figure 4 the cache usage by the test applications itself does not mimic the real application. When running the load regulator in parallel to the test application the hardware usage becomes almost identical. Additionally CPI has increased from 1.15 to 1.96 which is close to the original real world measurements of 2.04. We can conclude that introducing a load-generator together with an already present application-benchmarking suite improves the result. Figure 5 shows the test application behaviour as a function of time. As can be seen in the figure there is a lot of oscillation around the desired value. A better tuned or improved regulation algorithm can probably reduce these oscillations. We elaborate on this in Section 6.

4.2 Characteristics Evaluation in Daily Production Test Environment

We have evaluated our approach described in Section 2 on a real scenario where a software update was implemented for the platform. All applications running on the platform were left unchanged. The platform change relates to incorrect behaviour of cache handling in some rare circumstances. Before implementing the suggested solution it was suspected that it could affect the behavioural characteristics of the platform therefore leading to increased load when running the production system.

We have been using a signalling application together with a load generator to simulate the much larger and complex telecommunication system. The application creates two processes per core bouncing signals between them at a certain rate. As a behavioural characteristics we measure the signal turnaround time, which increases by 1,43% (average of 312 samples on 7 cores) with the software update running the signalling application alone.

When altering the load characteristics to mimic the cache usage for a real production system (with respect to L1 Instruction, L1 Data and L2 Data cache usage) the difference is much larger and easily detectable. With the software update, the signal turnaround time increases by 9,13% (average of 74 samples on 7 cores) compared to running without the software change. As a reference running the same software update on the complete production system the CPU load change is 7.57% (average of 606 samples on 6 cores).

We should be careful making a direct comparison between signal turnaround time and CPU load since they do not quantify the same metric. They should however be related since a higher CPU-load gives an increased signal turnaround time due to longer processing time at the sending and receiving processes. Measuring CPU load for the signalling application is not easily done since the cache-miss generators also cause some CPU load. For the production system signal turnaround time is not possible to measure since the production system lack this mechanism. We have not yet found a fully comparable metric that is easily measurable among both systems. Interpreting the figures from the evaluation

shows, in this particular case, a relationship between the CPU-load on the production system and signal turnaround-time on the model system. How to quantify this relationship is something that needs to be further studied.

5. RELATED WORK

Bell and John [3] describes a similar approach to ours. They define a method to model an application by synthesizing vital metrics. The model is then used to automatically create a representative test application with similar characteristics to the original one. They have applied this method on the SPEC2000 benchmark suite and the result shows that IPC differs on average 2.4% between the original applications and the model applications. Other metrics differ a degree slightly higher than ours, I-Cache 8.6% and L2 cache misses not explicitly written but to a large degree. Starting with the synthesizing procedure we use a feedback control loop to model the system while Bell and John [3] use statistical simulation with instruction traces, described by Nussbaum and Smith [10]. Bell and John states that the synthesis procedure is semi-automatic and an average of ten passes with some manual intervention is needed to tune the synthesis parameters. As a comparison feedback control allows the synthesis procedure to converge with no user interaction. Additionally, the model in our case is described by configuration parameters fed to a generic application. For Bell and John this is done at compile time requiring recompile to change its configuration. Another difference in our approaches is that we use a signalling application to detect any performance changes between releases while Bell and John uses IPC.

Joshi et al. [9] have formulated a concept called performance cloning that can be used to synthesize characteristics from a proprietary application and create a model that mimics a similar behaviour. In effect Joshi et al. implements a similar methodology as Bell and John in [3] but have refined the memory and branching model to be hardware agnostic.

Doucette and Fedorova [5] have implemented a similar functionality to ours when generating cache misses to determine application sensitiveness for different architectures. For example if an application is sensitive on one particular resource and another architecture has different amount of that resource the application performance is to some extent related to the hardware in the same way as the generator functions. One can in other words to some extent predict the performance of an application without actually running it on the target platform. As in Cache Pirating by Eklöv et al. [6] our application steals hardware-resources from other applications thus starving them. Our approach is to use the a cache miss generator to mimic a certain environment, while the cache pirate is used to reduce the available hardware-cache to determine the application demand for cache and memory bandwidth. We also work on a core-private cache instead of a shared cache. Saavedra and Smith [17] explain how to understand cache memory structure and how to generate misses, associativity etc. Alameldeen et al. [1] investigate server platforms and come to an interesting conclusion that it is quite difficult to create simulations of production systems. In their work they recreate the desired characteristics by using a tailored work-load suite. Our approach is similar but since they have shown some difficulties to recreate a similar hardware-load profile we use feedback-based load generator to achieve an approximation of the production application.

In the area of continuous system monitoring we can find interesting relations, such as Anderson et al. [2]. In their approach they implement a low intrusive (1%-3%) sample based mechanism to gather system wide information. The sampling is implemented by means of periodically executing sampling interrupts generated by performance counters. In our work this is done by a periodically executing process gathering performance counters in a ring buffer. One of the standard work when monitoring or measuring systems is the LM-Bench suite by Mcvoy and Staelin [15]. It is useful to measure and calculate cache and memory timings. Our platform is unfortunately not supported by the tool so a change to Linux was necessary to give insight into the characteristics. Regarding measurements with performance monitor counters Eranian [21] claims that they are a vital part in performance measurements and evaluation. Her investigation is done on a different hardware (x86) but the basic concept is the same; run a measurement application gathering information for later evaluation. In our case we have extended this idea to let the samples feedback into a regulation algorithm to produce cache misses.

6. FUTURE WORK

There are a number of issues that would be interesting to investigate further.

- To estimate characteristics for a production application running on hardware currently being designed we would like to create a model in the same way as being described in Section 2. Such estimation can be useful to make early performance discoveries reducing time-to-market for new hardware. The model needs to take into account for example cache size changes such as the square root method describes; doubling the cache reduces miss rate by $\sqrt{2}$, described by Hartstein et al. [14].
- Extending the simulation model with additional measurable metrics would improve the accuracy of the model, for example; L3 cache hit rate, Branch hit/miss rate, Interrupt rate. A more complete model would reduce the need for the signalling application, which is needed at present state to get representative result for complex systems.
- A more complete implementation of counters would, apart from giving a better approximation of the real system, also give deeper insight into the behaviour of the system. Investigations such as Eyerman et al. [8] where they investigate individual contributors to CPI would give interesting information about the characteristics of a particular system.
- The implementation described here generates bursts of memory accesses that are not representative to a real-world application. Such bursts may lead to congestion problems towards the bus and main memory. We don't yet know about the true behaviour of the real-world application so an improvement would be to investigate this further.
- To decrease the converging time and accuracy an improved regulation algorithm would be beneficial. The HW-properties we are regulating are connected to each other causing undesired side-effects. For example when one of the properties, such as L1 D cache, is changed it may cause a change for L2 D-cache. This cause

problems converging to the desired state. Using more advanced regulation algorithms may improve the behaviour.

- Reduce the sampling interval. In the current implementation this is set to 1 second per sampling item. This means that it takes several seconds between each sample. This causes the regulation algorithm to converge slowly. It is also difficult to observe transients since only the average over the sampling interval is recorded.
- Scale the regulation algorithm to a multi-core environment. Try different load profiles located on all cores on the CPU and see how that will influence application performance on other cores.

Acknowledgment

This work is funded by Ericsson AB and by the Swedish Knowledge Foundation (KK stiftelsen) through the ITS-EASY program.

7. REFERENCES

[1] A. R. Alameldeen, M. Martin, C. J. Mauer, K. E. Moore and M. Xu. Simulating a 2M$ Commercial Server on a 2K$ PC. *IEEE Computer*, pp. 50–57, Volume 36, Issue 2, Feb 2003

[2] J. Anderson, L. Berc and J. Dean. Continuous profiling: where have all the cycles gone?. In *ACM Transactions on Computer Systems*, pp. 357–390, Vol. 15, No. 4, November 1997

[3] R. Bell and L. K. John Improved automatic testcase synthesis for performance model validation. In *Proceedings of International Conference on Supercomputing (ICS)*, pp. 111–120, 2005.

[4] M. Bergqvist, J. Engblom, M. Patel and L. Lundegard. Some experience from the development of a simulator for a telecom cluster (CPPemu). In *Proceedings of the 10th International Association of Science and Technology for Development*, pp. 13–15, Nov. 2006.

[5] D. Doucette and A. Fedorova. Base vectors: A potential technique for microarchitectural classification of applications. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34*, 2007

[6] D. Eklöv, N. Nikoleris, D. Black-Schaffer and E. Hagersten. Cache Pirating: Measuring the Curse of the Shared Cache. In *Proceedings of Parallel Processing (ICPP), 2011 40th International Conference*, Sept. 2011.

[7] Ericsson. Ericsson unveils new products, partnerships and increased market share at mwc 2012. `www.ericsson.com/thecompany/press/releases/2012/02/1589097c`, 2012.

[8] S. Eyerman, L. Eeckhout, T. Karkhanis and J. E. Smith. A top-down approach to architecting CPI component performance counters. In *IEEE micro*, pp. 84–93, Vol 27, Jan-Feb, 2007.

[9] A. Joshi, L. Eeckhout, R. H. Bell Jr. and L. K. John Distilling the essence of proprietary workloads into miniature benchmarks. In *ACM Transactions on Architecture and Code Optimization*, pp. 1–33, Vol 5, 2008.

[10] S. Nussbaum and J.E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of Parallel Architectures and Compilation Techniques*, pp. 15–24, 2001

[11] Freescale. *P4080 Reference Manual*. Rev F. Ref f edition, 2009.

[12] Freescale. *e500mc Core Reference Manual*. Rev. f edition, 2010.

[13] Gartner. High Tech & Telecom Providers (HTTP). `www.gartner.com/technology/consulting/high-tech-telecom-providers.jsp`, 2012.

[14] A. Hartstein, V. Srinivasan and T. Puzak. Cache miss behavior: is it $\sqrt{2}$? In *Proceedings of the 3rd conference on Computing frontiers*, pp. 313–320, 2006.

[15] L. Mcvoy and C. Staelin. lmbench : Portable Tools for Performance Analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, p. 23, January 1996.

[16] K. Rowe. Time to market is a critical consideration. In *eetimes online paper*, `www.eetimes.com/discussion/guest-editor/4027610/Time-to-market-is-a-critical-consideration`, 2012.

[17] R.H. Saavedra and A.J. Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. In *IEEE Transactions on Computers*, pp. 1223–1235, October 1995.

[18] Stackoverflow. Generate Instruction Cache misses. In *Stackoverflow forum*, `stackoverflow.com/questions/9793660/what-are-the-causes-for-instruction-cache-miss`, 2012.

[19] P. Taylor. Battle lines are drawn for the future of 4G. In Financial Times online paper, `www.ft.com/intl/cms/s/0/399b1508-d9d8-11dc-bd4d-0000779fd2ac.html#axzz1va5rEtRx`, 2008.

[20] J. Scarpati. Faster time to market with next-gen OSS. `www.telecomasia.net/content/faster-time-market-next-gen-oss`, April 26, 2011

[21] S. Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on memory systems performance and correctness*, pp. 26-30, 2008