# xxx: A Proposal for a New Cache Monitoring Architecture

Martin Schulz, Jie Tao, Jürgen Jeitner, and Wolfgang Karl

Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik der Technischen Universität
München, Arcisstr. 21, D-80290 München, Germany

{schulzm,tao,jeitner,karlw}@in.tum.de.

March 29, 2002

## Abstract

**Keywords**
Hybrid Cache Monitoring, Memory Access Histograms

## 1 Motivation

It is a well known problem that the widening gap between memory and processor speed has a drastic impact on the overall performance of current systems. In order to tackle this problem, which is often refered to as the "Memory Wall" [**?**], modern architectures employ several caches in deep memory hierarchies. This allows to prevent a large percentage of memory accesses from having to directly access the main memory and thereby enables to hide memory latency.

In order to fully exploit the advantage of caches, however, applications need to exhibit a suitable memory patterns, which exploit both spatial and temporal locality and hence are capable of of reusing cache entries. In addition, potential cache line conflicts need to be taken into account and as far as possible avoided, a problem especially important in caches with low associativity. As a consequence this means that appliations need to be cache–aware and the programmers are responsible to write their applications accordingly.

As this task can be quite complex and requires extensive knowledge about both the memory hierarchy and the application's memory access characteristics, the programmers need to be able to rely on tools and/or monitoring techniques, which allow an appropriate analysis of the application on a specific target architecture and thereby enabled the necessary optimizations. Existing approaches in this area can be roughly divided into two classes: simulation systems covering the complete memory hierarchy and hardware counters built into the processor. While the first option provides very detailed information and also enables the examination of the access behavior in individual application fragments and to arbitrary segments in the process, it is not generally applicable, leads to large output data sets (often in the form of huge traces), is restricted to rather small test codes, and slow in its execution. The second option, the hardware counters as they are already present in most modern processor architectures, on the other hand allow precise and low–intrusive on–line measurements in hardware during the execution of applications, but are in turn restricted to very specific, mostly global events that can be monitored; often e.g. only the total number of cache misses or the number of memory accesses can be acquired. This information is therefore often not sufficient for a detailed optimization.

The work presented in this paper proposes a new cache monitoring approach, which aims at combining the advantages of these two techniques within a single system. The core is a novel type of a hardware monitoring facility with the same positive properties regarding speed, intrusiveness, and applicability as given with traditional counters, but at the same provides a level of detail currently only available from simulation systems. It is based on memory access histograms which show the number of accesses at all levels of the memory hierarchy with respect to the individual target addresses and hence allow an address specific observation and optimization. These histograms are directly compiled by the proposed hardware mechanism by observing the memory traffic on all levels of the memory hierarchy and by efficient pre–filtering of this information in a dynamic counter array. This guarantees that the hardware monitor can rely on all memory events taking place in the system without incurring huge overheads.

This monitoring principle is based on an earlier work undertaken in the area of monitoring NUMA interconnection networks in loosely coupled shared memory systems [**?**]. This particular monitor is also based on the principle of snooping a bus, in this case the bus connecting a single node to the actual interconnection fabric, and has been developed all the way to a hardware prototype. It has shown that the concept is feasable and can be implemented without major impact on the performance of the host system and with modest hardware complexity [**?**].

The new work presented here builds on top of this idea and shows how this monitor concept can be extended into the realm of memory interconnection fabrics in a straightforward way. As

this, due to the tight integration of modern cache architectures into the processor core, would require modifications of the processor core and hence is not feasible for an academic project, a preliminary stuy is presented based on a comprehensive simulation environment called SIMT [**?**] which has been adapted for this purpuse. First experiments show ...

The remainder of this paper is structured as follows. Section 2 introduces the problem of cache monitoring in more details and discusses some existing approaches. The new xxx approach will then be presented in Secion 3 followed by first experimental results using a comprehensive simulation framework in Section 4. The paper is then rounded up by an outlook on future work in Section 5 and by some concluding remarks in Section 6.

# 2 Caching and its Problems

In the last decades both processor and memory speedup have been growing at an exponential rate. However, the growth rate of the memory speedup has been significantly lower than the one of that could be observed for processor speeds [**?**], leading to a significant and continuesly growing gap. This problem, often reffered to as the memory wall [**?**], has a severe impact on modern micro processors, as they can not fully exploit their potentials as the memory access acts as a severe bottlenecks and prohibts a sufficient stream of data and instructions.

Caches, which have been introduced to compensate for this problem and act as transparent fast buffers for repeatedly used data, are able to mask this bottleneck to a certain degree. Due to this, basically any modern computer systems currently relies on caches, often even organized as several layers of caches in deep memory hierarchies. They can, however, only be effective if the data stored in them is actually reused by the running applications. This can be achieved through various well-known techniques (E.g. [**?**]) which improve both the spatial and temporal locality of the memory access behavior of the target application and thereby make the applications explicitly cache–aware. This process, however, requires extensive knowledge about both the application memory access pattern and the underlying architectural specifics (including cache size, cache line size, and associativity), as well as the interaction between these two issues.

As this knowledge in most cases can not simply be gained by a simple, manual source code analysis, it is mandatory to provide the programmer with appropriate tools which allow the observation and evaluation of the behavior of an application in a given architectural environment. This has lead to mainly two approaches which are currently used: the full simulation of applications and caches and the inclusion of hardware counters into processors.

The increasing complexity of compute systems has led to a wide use of simulation tools for predicting and evaluating performance on target architectures. Among the developed systems which usually target the complete architecture, several simulators focus on the cache performance, including the locality issues on uniprocessor systems . These simulation based approaches generally rely on the tracing of references, which is either stored in a trace file or directly transfered to the cache simulator, to gather statistical data about cache behavior. This data can be used to analyze the cause of cache misses and further to optimize the data layout towards a better data locality. For large applications, however, such tracing can significantly slow down the execution due to the huge amount of memory operations. This situation will be worse when a trace file is used.

MS: Counters - pro and cons

why this is not sufficient, where are the problems

what is needed - requirements: accuracy, address relation, no overhead, ... anything else?

# 3 Cache Monitoring based on Memory Access Histograms

This work introduces a new hardware–based approach for cache monitoring, which follows these guidelines and requirements. The result is a novel monitoring architecture which delivers fine–grain spatial information of the complete memory access behavior of an application in an on-line fashion with a minimal intrusion overhead. This can then be used as the basis for any kind of cache optimization of the monitored application.

## 3.1 Cache–like Counter Organization

The core of the proposed system is an associative counter array, which is used to count the occurance of memory events. This events are detected by snooping the memory bus under investigation, e.g. the link between the L1 and the L2 cache showing all L1 cache misses. From this bus the address of the event is extracted and used as an index to select the corresponding counter in the counter array. This counter is then incremented on–the–fly with influencing the system under investigation.

This principle, however, would require a counter for each potential memory address that can occur on the particular memory bus — a requirement which is certainly infeasible. The proposed approach, therefore, implements a cache–like logic for the counter array. It comprises a limited number of counters which are mapped to addresses in a fully associative manner in the order of their occurance on the bus. Once all counters have been allocated and are in use, new requests for counters are satisfied by evicting counters from the counter array into a special ring buffer in main memory. The freed counter can then be reset to 0 and reallocated to count events for a new address.

This ring buffer, therefore, contains incomplete counting results which can then be combined by software to get the final memory access historgram. This post–prcessing is very light–weight since it only requires the summation of the partial counting results and could therefore even be done periodically, e.g.

during an idle loop or the invocation of the scheduler, without a significant impact of the overall performance.

In addition, this dynamic swapping technique also allows to keep the number of bits used for each counter at a reasonable level, as full counters can also be swapped out into the ring buffer and counted as partial results. The counter can then be reset to 0 and then again start counting for the particular address.

In summary, this approach therefore provides a beneficial tradeoff between the ability to individually count events in relation to their addresses and a managable hardware complexity. It also keeps the impact on the overall system, caused by the swap out events and the associated post–processing, at an acceptable level guaranteeing a low intrusion on the running application. The approach requires, however, a direct access to the individual memory paths resulting in the need for an implementation within the processor package in order to be effective. It can therefore be seen as alternative to replace traditional counters within the processor with a similar integration complexity, but with a significantly higher–level functionality.

## 3.2 User-definable Granularity

Despite the complexity reducing technique discussed above, the finest possible granularity can still lead to rather large histograms, even if the user does not need this fine granularity to the full extend. The presented monitoring concept, therefore, includes the ability to influence the granularity of the monitored data and thereby to adjust the monitor behavior to the intended use of the data.

This granularity control is thereby done in two steps. First a mask is applied to all monitored events allowing to delete unimportrant lower bits from any address. This, e.g. allows in an easy way to restrict the monitoring to full cache line, which normally anyway represent the smallest unit of transfer on the memory bus.

Besides this simple mask, the monitor also includes a more sophisticated, dynamic granularity control. This allows the aggregation of neighboring events during the monitoring process without predefined boundaries. This aggregation is done on–the–fly and can be controled by a user–definable parameter specifying the maximal range of addresses which are allowed to be combined.

## 3.3 The Need for Multilevel Monitoring

The approach discussed so far allows the monitoring of a single location within the processor (e.g. between L1 and L2 cache showing the L1 misses). While this provides a good first insight into the cache behavior of an application, it often does not deliver the information to fully judge the relevance of the observed behavior. This stems from the fact that such single monitor is not capable of providing any relation of the observed behavior to the overall global behavior on the complete memory hierarchy and hence e.g. also misses information like miss rates.

In order to compensate this deficit, it is necessary to deploy multiple monitors, one on each level of the memory hierarchy (see also Figure 1). Each monitor (three in case of a typical 2–level hierarchy) computes the independent memory access histgrams occuring at the corresponding locations in the system. These individual histograms can then be combined into a final result, which shows the combined memory access information.

Based on this final histogram, it is then possible to relate numbers of hits or misses in any cache and on any address segment to the total number of accesses on that memory location. This can be used to compute both hit or miss rates for any cache with respect to given address regions. Combined with higher–level tool or debugger support, this can then be mapped back to source data structures allowing an easy detection of memory bottlenecks within applications.

## 3.4 Adding Temporal Information

The main drawback of this approach that it, while delivering results with respect to a fine–grained spatial resolution, does not provide any temporal information of the memory access behavior. Histograms, as they are deployed here, only contain the aggregated memory behavior of the whole monitoring period.

For many codes and application scenarios, however, some temporal information would be useful for the evaluation of the monitored information. Especially, distinct phases within the computation should be separatable as they often also exhibit different memory access patterns which should be examined separately. This includes distint areas for startup, preprocessing, and psotprocessing as well as phases within the computation itself, e.g. in the case of iterative methods. This can be achieved by so–called monitor barriers, which trigger a complete swap out of all partial results contained within the counter array followed by a full reset. This allows the complete the histogram of the phase before the barrier and re–initializes the monitor for the next phase.

In addition, to these rather course grain temporal phases, also other, more fine–grain temporal separation techniques could be used. One option would be to separate function invocations by introducing monitor barriers around call sites. Together with an appropriate software infrastrcuture, it would then be possible maintain several independent ring buffers for each monitor allowing the separate recording of individual historgrams, which e.g. can show the aggregated memory behavior of all invocations of specified functions during the application runtime.

This technique should, however, only be used with care as these monitor barriers incur a certain amount of overhead. They lead to the swapping of all monitor counters and their transfer to main memory and also infer a certain startup overhead in the new phase. This can be solved, if only a single function or type of region is monitored; in this case the monitor barrier can be replaced with a single switch which enables or disables the monitor at the corresponding code locations. This avoids any overhead, but still provides the intended selective information.

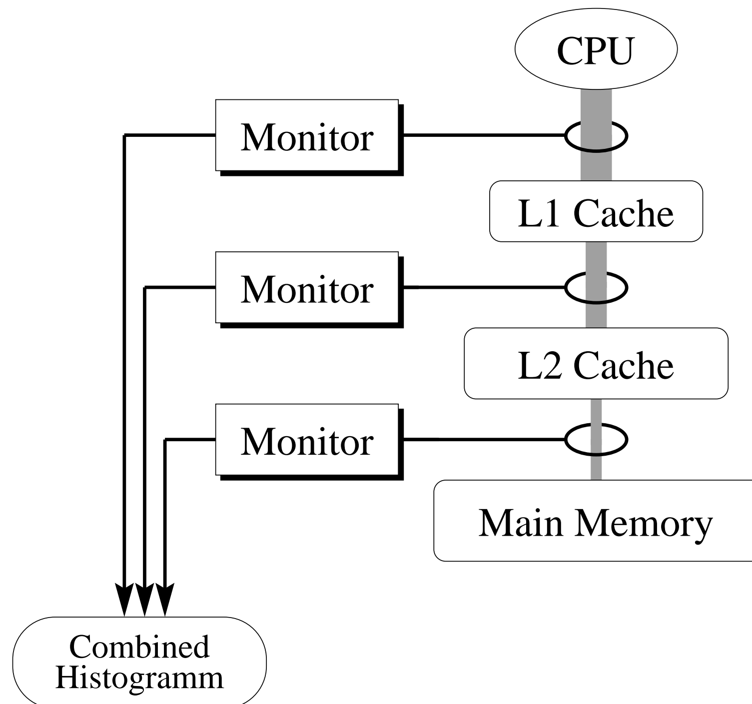


Figure 1: Multilayer monitoring architecture across complete memory hierarchy.

A second problem associated with temporal fine–grain cache monitoring, however, can not be avoided: the selective monitoring of certain regions will always to some degree reflect and depend on the memory behavior of the code regions touched before. These are responsible for the cache contents at the beginning of the region and hence impacts the memory behavior of the monitored part. Therefore, the monitoring of temporally fine-grain regions will be greatly distorted and it is even debatable whether this information is useful at all.

# 4 Experimental Results

As mentioned above, the presented monitoring approach assumes an implementation with the processor package in order to get direct access to the various memory busses. As this is infeasible for an academic project, the studies for this work are currently executed on top of an accurate and comprehensive simulation environment. This simulation , however, covers the complete behavior of the proposed monitor and hence gives a realistic representation of the expected hardware behavior.

## 4.1 Experimental Setup using SIMT

The multilayer hardware monitors have been simulated within a multiprocessor simulator called SIMT [1]. SIMT is a general tool designed for evaluation of clusters with NUMA organizations, with a focus of the memory hierarchy which comprises an L1 cache, an L2 cache, and a global virtual memory system. For cache simulation SIMT uses a flexible approach, where all parameters related to cache configurations, e.g. cache size, associativities, and read/write policies, can be specified through command line options.

Based on this cache simulator, the hardware monitors can be simulated. They capture the cache events, which are generated by the cache simulator, extract information of interest from them, and record the information in the counters and further in the ring buffers. The monitoring data is then processed by a low-level software, with a result of an access histogram in the form of accesses to each component of the memory hierarchy. This enables an initial study of the proposed approach.

The following experiments have been conducted using several codes taken from the SPLASH-II suite [**?**]. These codes, which where originally intended for parallel benchmarking in shared memory machines, offer a wide spectrum of memory access patterns. The concrete codes used in this study together with the deployed dataset sizes are shown in Table 1.

## 4.2 Global Monitoring

MS: global results for a view codes and their relation to the source, done with the simple simulator, several codes

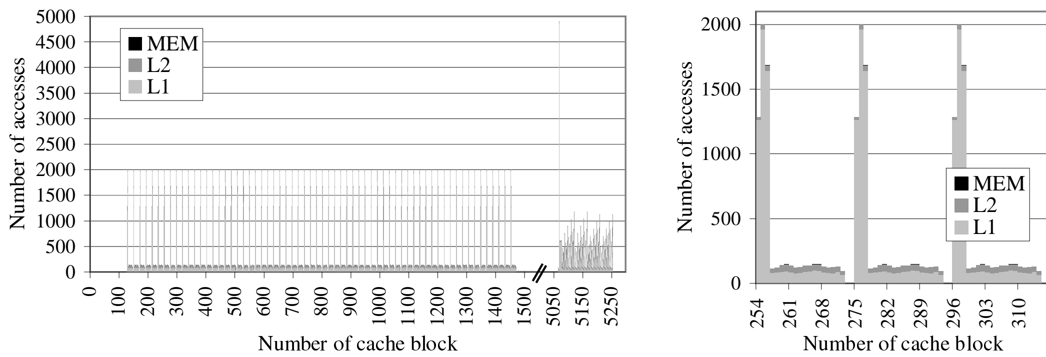


Figure 2: Accumulated Memory Access Histograms of WATER — complete address space (left) and zoom on range for 3 molecules (right).

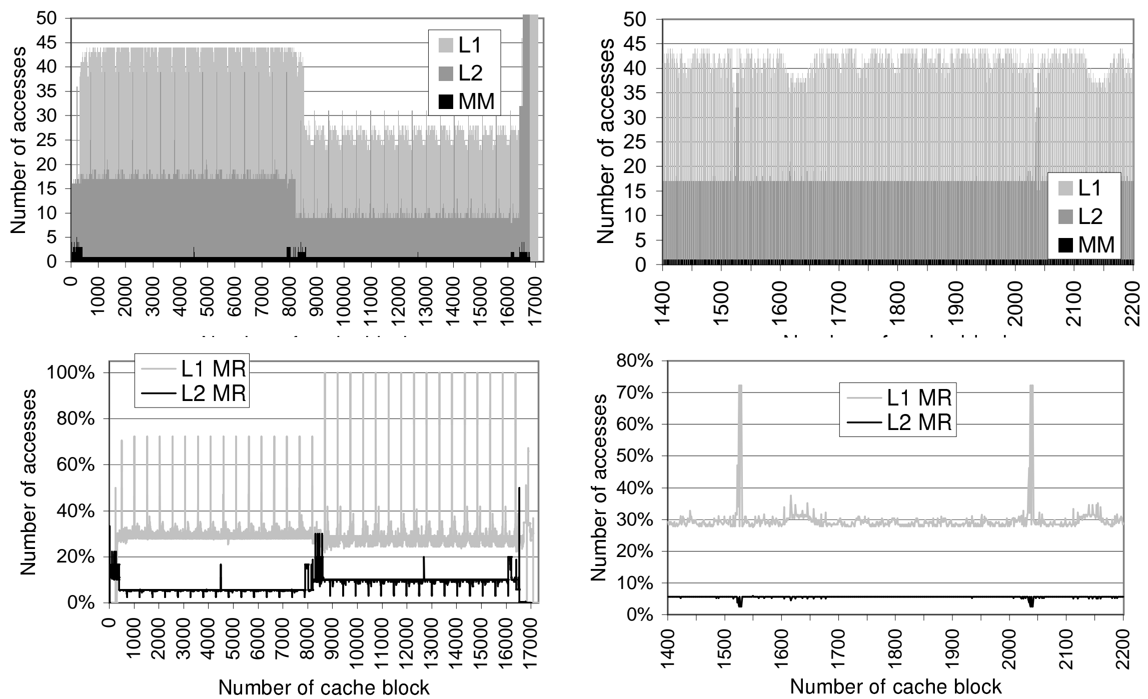


Figure 3: Non–accumulated Memory Access Histograms of RADIX, complete address space (top left) and zoom into first array (top right) — Cache miss rate, complete address space (bottom left) and zoom into first array (bottom right).

| Code | Data set size |
|---|---|
| LU decomposition | Dense matrix of 128x128 |
| RADIX sort | 65538 keys |
| WATER | 64 molecules |

Table 1: Test codes and their data set sizes.

### 4.3 Phase–wise Monitoring

As mentioned above, it is in many cases useful to introduce some notion of temporal behavior and provide monitor with respect to that. This is especially useful if the code exhibits distinct phases, as it e.g. the case for the LU decomposition. This code is able, given a dense input matrix $A$, to compute a lower and an upper triangular matrix, $L$ and $U$ respectively, in a way that $A = L * U$ and hence can be used to solve a linear system of equations in the form of $A * x = b$.

The code used for this experiment, which was taken from the SPLASH-II suite [?], performs this computation after subdividing the input matrix in subblocks (in this case in blocks of 16x16 elements) and computes on these subblocks in several steps. In each step one block on the diagonal of the matrix is used for the core computation; however, also other blocks are updated resulting in a complex memory access pattern. Nevertheless, the block structure is still clearly visisble in the access histograms, as can be seen below.

As the code has already been parallelised for a shared memory environment, it already contains an implicit separation into phases through the included shared memory synchronization barriers. In total, the code with the utilized dataset executes in 19 distinct phases. Figure 4 shows the monitored memory access histograms, again in an accumulated fashion, for four of them, followed by an full histogram over the complete execution time. As expected, it can be seen that the complete histogram is simply a sum of the histograms produced in the individual phases.

Besides that, it can be seen that the LU code operates with two different types of phases (phase 3 and 5 vs. 4 and 6), which now, after the separation into phases, are clearly visible and can be analyzed separately. The first type is a preparation step with a high locality on very few memory blocks (mainly the block on the diagonal and to a less degree the blocks in the same row and column). The second phase, on the other hand, does not show similar locality behavior, but rather updates a large range of blocks in a very regular fashion.

In summary, the separation of the monitored access histograms into phases strongly complements the ability of fine–grain monitoring given with the memory access histograms. Combined, this allows the in–depth analysis of several distinct memory access patterns within a single code.

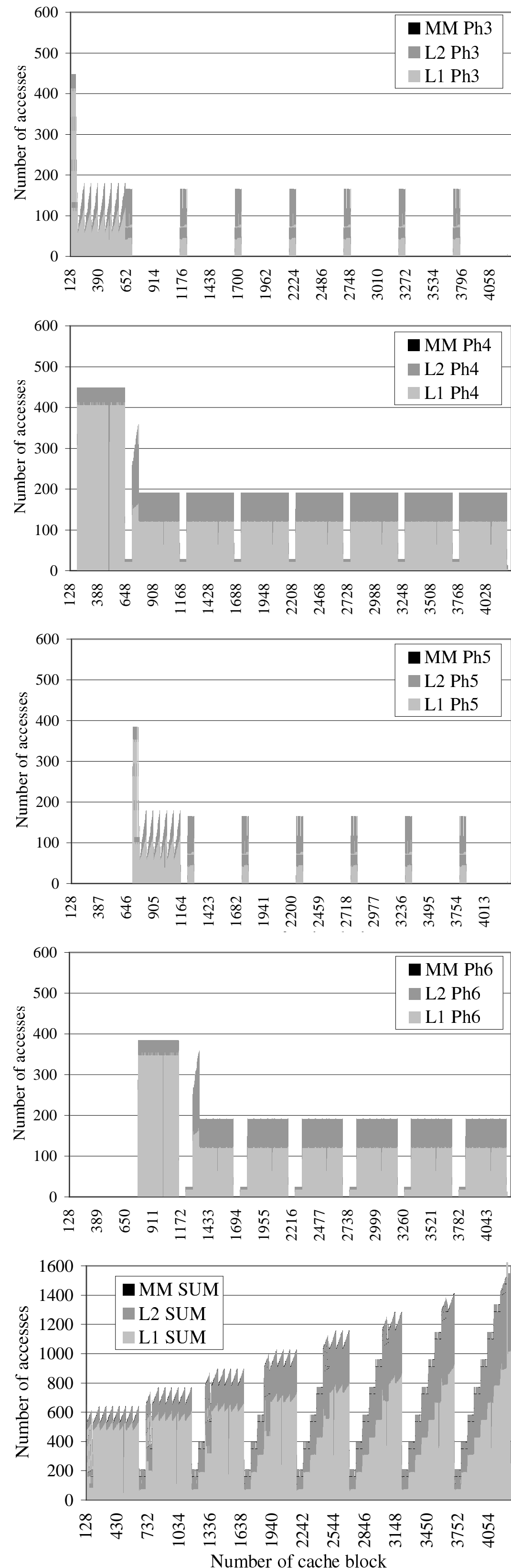


Figure 4: Accumulated Memory Access Histograms various of LU Phases.

### 4.4 Tradeoff between Precision, Hardware Complexity, and Runtime Overhead

All histograms shown so far have been acquired at highest possible accuracy by assuming cache line granularity. This however, can lead to a visible runtime overhead due to the on–line delivery of the memory access histogram. In order to compensate for this and to keep the influence of the monitor on the application execution to a minimum, two options can be explored: the reduction in granularity which will lead to a reduction in accuracy or an increased number of counters in the associative counter array which will increase the hardware complexity of the monitoring system.

In order to evaluate this further a series experiments, again using the LU code, have been conducted in which these parameters have been varied: the granularity from a single cache line to aggregated counters for six contiguous cache lines and the number of counters present in the monitor from 16 to 128. In all cases, only the events from the monitor before the L1 cache have been used, as this is the monitor which has to deal with the by far largest number of events within a monitoring hierarchy. Table 2 shows the results in terms of error in accuracy and number of swap–out events, each event leading to the transfer of one counter and its associated address tag. The error in this table has been computed in relation to a follow accurate histograms taken with granularity 1 using the following formula:

$$Error = \frac{\sum_{addr=0}^{max.addr} |Access_a cc(addr) - Access_n onacc(addr)|}{\sum_{addr=0}^{max.addr} Access_a cc(addr)}$$

## 5 Extensions and Potentials

This work, as presented here, is currently in an early stage and hence certainly not yet fully explored. Besides the obvious tasks, which include a concrete estimation of the hardware cost based on a full VHDL model, further simulations of the overall system impact of the monitor swap activities, and further detailed experiments using the current simulation system, we will also explore a series of possible extensions.

### 5.1 Observation of Other Events

Besides the actual memory accesses discussed so far, this kind of monitoring system is capable of observing any kind of event that can be related to an address. This includes cache line state transitions like cache line loads and flushes, as well as state transitions used by cache coherency protocol in multiprocessor systems, like SMPs. For the latter case especially the commonly used MESI protocol [?] is of interest, as the hardware monitoring architecture would be able to register the change in the MESI states and hence enable the detection of exclusively used memory regions and regions with heavy sharing between processors.

### 5.2 Memory Bank Monitoring

The current approach is solely intended for observing transaction within the memory hierarchy and for tracking cache problems. However, also other bottlenecks in the memory system can exits with also a severe impact on the overall performance. One concrete example are interleaved memory banks, which can only be exploited to their full extend if applications are accessing these banks appropriately and with a minimum of conflicts — a problem quite similar to cache–aware programming.

### 5.3 Graphical Tool Support

The information acquired through this kind of by its nature still quite low–level, mostly based on monitored physical addresses, and hence not easily understandable by any programmer or user. Therefore, appropriate visualizations need to be established which translates this information to the level of abstraction of the source code. While the visualizations shown in the previous sections are currently hand–crafted (a method good enough for a first evaluation, but certainly not for suitable for production use), the next step will be to develop appropriate high–level visuatlization tools.

## 6 Conclusions

The development of cache–aware applications is a crucial issue for the exploitation of the potential in modern computer and memory architectures. This, however, is a complex and cumbersome task and hence needs to be supported by appropriate tools and/or hardware monitoring techniques. Current solutions either rely on simple hardware counters, which only give a global overview of few events, or on simulation, which is both time-consuming and uncapable of dealing with large, realistic workloads.

In this work a novel hardware cache monitoring architecture has been presented which can be intregrated into the memory hierarchy without major overhead or hardware complexity and at the same time enables an address related monitoring based on memory access historgram, currently only known from software simulation techniques. This is achieved by an observation of the memory traffic on all levels of the memory hierachy using on adaptive counter mechanism. This is designed in a way that the data that is generated by the monitor is minimized. In summary, this allows the accurate evaluation of the cache behavior of full–size applications in realistic scenarios, and as a consequence enables specifically targeted optimizations in areas which exhibit a bad cache behavior.

This concept has been demonstrated using several benchmark codes from the SPLASH-2 suite [?]. The result was a full memory access histogram showing all levels of the memory hierarchy and which allowed a precise overview of the application's behavior. In addition, the experiments have shown that the precision of

| | C=16 | C=32 | C=64 | C=128 |
|---|---|---|---|---|
| G=1 | 0.00% | 0.00% | 0.00% | 0.00% |
| G=2 | 2.05% | 2.05% | 2.05% | 2.05% |
| G=3 | 4.61% | 4.61% | 4.61% | 4.61% |
| G=4 | 4.97% | 4.97% | 4.97% | 4.97% |
| G=5 | 6.34% | 6.34% | 6.34% | 6.34% |
| G=6 | 6.56% | 6.56% | 6.56% | 6.56% |

| | C=16 | C=32 | C=64 | C=128 |
|---|---|---|---|---|
| G=1 | 1052893 | 702290 | 582964 | 491489 |
| G=2 | 423823 | 342031 | 294309 | 89604 |
| G=3 | 423823 | 29148 | 197080 | 45797 |
| G=4 | 246218 | 195801 | 73025 | 27952 |
| G=5 | 212598 | 163740 | 53182 | 21731 |
| G=6 | 158171 | 119215 | 29148 | 18071 |

Table 2: Error in counter accuracy (left) and number of swap–out events (right) for various number of counters (C) and granularities (G) using the LU code.

these access histograms is close to a full simulation, while keeping the amount of necessary data transfer and hence the system overhead to a minimum. This ensure the applicability of the approach ...

# References


[1] J. Tao, W. Karl, and M. Schulz. Using Simulation to Understand the Data Layout of Programs. In *Proceedings of the IASTED International Conference on Applied Simulation and Modeling (ASM 2001)*, pages 349–354, Marbella, Spain, September 2001.