Predicting Cache Contention with Setvectors

Michael Zwick, Florian Obermeier and Klaus Diepold *

Abstract—**In this paper, we present a new method called *setvectors* to predict cache contention introduced by co-scheduled applications on a multicore processor system. Additionally, we propose a new metric to compare cache contention prediction methods. Applying this metric, we demonstrate that our *setvector* method predicts cache contention with about the same accuracy as the most accurate state-of-the-art method. However, our method executes nearly 4000 times as fast.**

Keywords: Setvectors, Coscheduling, Cache contention.

1 Introduction

With multicore processors, chip manufacturers try to satisfy the ever increasing demand for computational power by parallelization on thread or process basis, making performance of computer systems more and more independent from the saturated processor clock speed. However, one important limitation that does not rely on processor clock speed, but on the computational power of the processor, is the ever increasing processor memory gap: Although both, processor and DRAM performance, grow exponentially over time, the performance difference between processor and DRAM grows exponentially, too. This happens due to the fact that"the exponent for processors is substantially larger than that for DRAMs" [7] and "the difference between diverging exponentials also grows exponentially" [7].

A way to deal with the exponentially diverging memory gap is to transform computational performance into memory hierarchy performance, making memory performance not only benefit from improvements of the memory hierarchy system, but also from better (and in a much higher rate evolving) processor technology. One possibility therefore is to spend computational power to find good application co-schedules that minimize overall cache contention. Reducing DRAM accesses by optimizing cache performance is a key issue in todays and tomorrows computer architectures.

L2 cache performance has been identified as a most crucial factor regarding overall performance degradation in multicore processors [2]. Figure 1 shows the effect of L2 cache contention on the SPEC2006 benchmark *milc*, running on core c0 of a dual core processor, when co-scheduled with applications *astar*, *gcc*, *bzip2*, *gobmk* and *lbm* on core c1. It can easily be seen that the performance of *milc* heavily degrades when co-scheduled with the *lbm* benchmark; other co-schedules however, have a much lower performance burden.

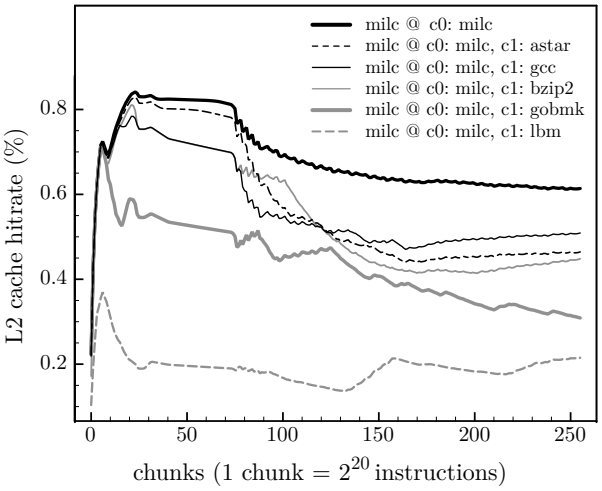

Figure 1: L2 cache hitrate degradation for the *milc* SPEC2006 benchmark when co-scheduled with different applications.

A requirement in order to optimize co-schedules for cache contention is a good metric to predict cache contention of application co-schedules from specific application characteristics. Although a number of methods have been investigated that predict L2 cache performance from some application characteristics for single core processors, so far only little effort has been spent to predict L2 cache performance of co-scheduled applications in a multicore scenario.

In this paper, we propose a new method called *setvectors* to predict cache contention in multicore processors. We compare our method to the *activity vectors* proposed by Settle et al. [6] and the circular sequence based *prob* model presented by Chandra et al. [1]. We show that our *setvector* method predicts optimal co-schedules with about the same accuracy of the best performing circular sequence based method, but, on average, executes about 4000 times faster.

*Lehrstuhl für Datenverarbeitung, Technische Universität München, 80333 München, Germany

The remainder of this paper is organized as follows: Section 2 presents state-of-the-art techniques to predict cache contention; section 3 introduces our *setvector* method. In section 4, we propose a new metric called *MRD* (mean ranking difference) to compare cache contention prediction techniques and discuss the parameters applied to our simulation. In section 5, we present our results. Section 6 concludes this paper.

2 State-of-the-art Cache Contention Prediction Techniques

In this section, we describe state-of-the-art techniques to predict cache contention in multiprocessor systems, namly Alex Settle et al.'s *activity vectors* [6] and Dhruba Chandra et al.'s *stack distance* based *FOA* (frequency of access) and *SDC* (stack distance competition) model [1] and their *circular sequence* based *Prob* (probability) model [1].

2.1 Settle et al.'s Activity Vectors

Alex Settle et al. studied processor cache activity and observed that "program behavior changes not only temporally, but also spatially with some regions hosting the majority of the overall cache activity."[6] To exploit spatial behavior of cache activity to estimate cache contention, they divide the cache address space into groups of 32 so-called *super-sets* and count the number of accesses to each such super set. If, in a given time interval, the accesses to a super set exceed a predefined threshold, a corresponding bit in the so-called *activity vector* is set to mark that super set as active.

To predict the optimal co-schedule B, C or D for a thread A, every bit in the activity vector of A is logically AND-ed with the corresponding bit in each B, C and D. The bits resulting from that operation are summed up for each thread combination $A \leftrightarrow B$, $A \leftrightarrow C$ and $A \leftrightarrow D$. As a co-schedule for A, that thread in {B, C, D} is chosen that yields the least resulting sum. [6]

2.2 Chandra et al.'s Stack Distance Based FOA and SDC Methods

In [1], Dhruba Chandra et al. propose to use *stack distances* to predict cache contention of co-scheduled tasks. Stack distances have originally been introduced by Mattson et al. [5] in 1970 to assist in the design of efficient storage hierarchies in virtual memory systems. In [3], Mark D. Hill and Allan J. Smith showed that they can also be easily applied to evaluate cache memory systems.

The method assumes a cache with LRU (least recently used) replacement policy and works as follows: Given a cache with associativity α, the number of $\alpha + 1$ counters $C_1, \ldots C_{\alpha+1}$ have to be provided *for each cache set* to track the reuse behavior of cache lines. If, on a cache access, the cache line resides on position p of the LRU stack, counter C_p of the corresponding cache set is increased by one. If the cache access results in a miss, i.e. if the cache line has no corresponding entry on the LRU stack (and therefore the cache line does not reside in the cache), then counter $C_{\alpha+1}$ is increased. This procedure leads to a so-called *stack distance profile*, as it is depicted in figure 2. The stack distance profile characterizes the positions of cache lines on the LRU stack when accessing cache data.

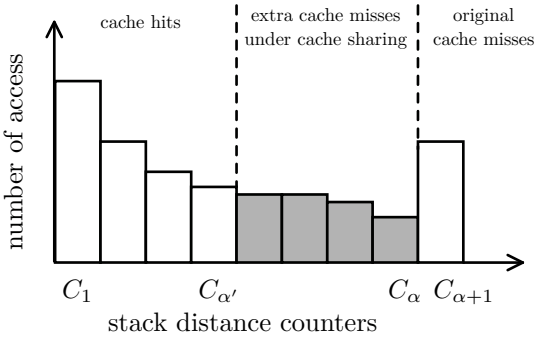

Figure 2: Stack distance histogram.

Given a stack distance profile, the total number of accesses to a specific cache set can simply be determined by summing up all C_i according to

$$accesses = \sum_{i=1}^{\alpha+1} C_i \tag{1}$$

and the cache miss rate can be calculated by

$$P_{\text{miss}} = \frac{C_{\alpha+1}}{\sum_{i=1}^{\alpha+1} C_i}. \tag{2}$$

For a smaller cache with lower associativity α', the miss rate can be computed as

$$P_{\text{miss}}(\alpha') = \frac{C_{\alpha+1} + \sum_{i=\alpha'}^{\alpha} C_i}{\sum_{i=1}^{\alpha+1} C_i}. \tag{3}$$

Chandra et al. exploit this equation to predict the cache miss rate under cache sharing. They estimate the *effective associativity* α' of a task when sharing the cache with another task according to

$$\alpha' = \frac{effCacheSize_x}{numCacheSets}, \tag{4}$$

where *numCacheSets* denotes the number of sets the cache is composed of and $effCacheSize_x$ the effective cache size that is available for thread x.

Within their *FOA* model, they calculate the effective cache size according to

$$effCacheSize_x = \frac{\sum_{i=1}^{\alpha+1} C_{i,x}}{\sum_{y=1}^{N} \sum_{i=1}^{\alpha+1} C_{i,y}} \cdot CacheSize. \tag{5}$$

Within their *SDC* model, they create a new stack distance profile by merging individual stack distance profiles to one profile and determine the effective cache space for each thread "proportionally to the number of stack distance counters that are included in the merged profile."[1]

The shaded region in figure 2 shows how the effective cache size is reduced by cache sharing.

While the *FOA* and the *SDC* model both are heuristic models, Chandra et al. also developed an inductive probability model that is based on circular sequences rather than on stack distances.

2.3 Chandra et al.'s Circular Sequence Based Prob Method

Circular sequences are an extension to stack distances in that they do not only take into account the number of accesses to the different positions on the LRU stack, but also the number of cache accesses between accesses to equal positions on the LRU stack.

Therefore, Chandra et al. define a *sequence* $seq_x(d_x, n_x)$ as "a series of n_x cache accesses to d_x distinct line addresses by thread x, where all the accesses map to the same cache set" [1] and a *circular sequence* $cseq(d_x, n_x)$ as a sequence $seq_x(d_x, n_x)$ "where the first and the last accesses are to the same line and there are no other accesses to that address" [1]. Circular sequences can be regarded as stack distances that have each counter C augmented with an additional vector n to hold a histogram of accesses for each distance. Figure 3 illustrates the relationship between sequences and circular sequences when accessing cache lines A, B, C and D.

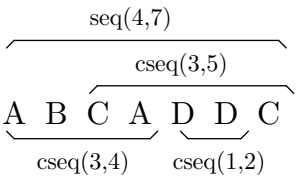

Figure 3: Relationship between sequences and circular sequences. A, B, C and D depict different cache lines.

For their circular sequence based *Prob* model, Chandra et al. compute the number of cache misses for a thread x when sharing the cache with an additional thread y by adding to the stand-alone cache misses $C_{\alpha+1}$ the values of the other counters $C_1 \ldots C_\alpha$, each multiplied with the probability that the corresponding circular sequences $cseq(d_x, \overline{n_x})$ will become a miss, where $\overline{n_x}$ corresponds to the estimated *mean* n for a specific d:

$$miss_x = C_{\alpha+1} + \sum_{d_x=1}^{\alpha} P_{miss}(cseq_x(d_x, \overline{n_x})) \times C_{d_x} \quad (6)$$

Chandra et al. calculate the probability that the circular sequence $cseq(d_x, \overline{n_x}))$ will become a miss by summing up the probabilities that there are sequences $seq_y(d_y, E(n_y))$ in thread y with $\alpha - d_x + 1 \leq d_y \leq E(n_y)$, where $E(n_y)$ represents the expected value of n in the thread y:

$$P_{miss}(cseq_x(d_x, \overline{n_x})) = \sum_{d_y=\alpha-d_x+1}^{E(n_y)} P(seq_y(d_y, E(n_y))) \quad (7)$$

$E(n_y)$ is estimated by scaling $\overline{n_x}$ proportionally to the ratio of accesses of y and x:

$$E(n_y) = \frac{\sum_{i=1}^{\alpha+1} C_{i_y}}{\sum_{i=1}^{\alpha+1} C_{i_x}} \cdot \overline{n_x} \quad (8)$$

The probability of sequences $P(seq_y(d_y, E(n_y)))$, in short $P(seq(d, n))$, is calculated recursively according to

$$P(seq(d,n)) = \begin{cases} 1 & \textbf{if } n = d = 1 \\ P((d-1)^+) \times P(seq(d-1, d-1)) & \textbf{if } n = d > 1 \\ P(1^-) \times P(seq(1, n-1)) & \textbf{if } n > d = 1 \\ P(d^-) \times P(seq(d, n-1)) + & \\ \quad P((d-1)^+) \times P(seq(d-1, n-1) & \textbf{if } n > d > 1 \end{cases}$$

where $P(d^-) = \sum_{i=1}^{d} P(cseq(i, *))$ and $P(d^+) = 1 - P(d^-)$ (cgf. [1]) and the asterisk $(*)$ in $cseq(i, *)$ denotes all possible values.

3 Setvector Based Cache Contention Prediction

In this section, we describe our setvector method. First, we present the algorithm to obtain setvectors. Second, we show how setvectors can be used to predict cache contention.

3.1 Generating Setvectors

Setvectors are composed of cache set access frequencies **a** and the number of different cache lines **d** referenced within a specific amount of time, typically about an operating system's timeslice. Within this paper, we collect one setvector for every interval at 2^{20} instructions. According to our proposal in [9] where we presented setvectors to predict L2 cache performance of stand-alone applications, we assume an L2 cache with 32 bit address length that uses b bits to code the *byte offset*, s bits to code the selection of the *cache set* and $k = 32 - s - b$ bits to code the *key* that has to be compared to the tags stored in the tag RAM. The setvectors are gathered as follows:

For every interval i of 2^{20} instructions do:

- First, set the 1×2^s sized vectors **a** and **d** to **0**.
- Second, for every memory reference in the current interval, do:

- Extract the set number from the address, e.g. by shifting the address k bits to the left and then unsigned-shifting the result $k + b$ bits to the right.
- Extract the key from the address, e.g. by unsigned shifting the address $s+b$ bits to the right.
- Increase **a**[set number].
- In the list of the given set, determine whether the given key is already present.
- If the key is already present, do nothing and proceed with the next address.
- If the key is not in the list yet, add the key and increase **d**[set number].

We end up with two 1×2^s dimensional vectors **a** and **d**. At index j, **a** holds the number of references to set j and **d** holds the number of memory references that map to set j, but provide a different key.

- In a third step, subtract the cache associativity α from each element in **d** and store the result in $\mathbf{d}'$. If the result gets negative, store 0 instead.
- In a forth step, multiply each element of **a** with the corresponding element in $\mathbf{d}'$ and store the result in the 1×2^s dimensional setvector $\mathbf{s}_i$.
- Finally, add $\mathbf{s}_i$ as the ith column of matrix **S** that holds in each column i the setvector for interval i.

Process next interval.

3.2 Predicting Cache Contention with Setvectors

The compatibility of two threads for a time interval i can easily be predicted by just extracting $\mathbf{s}_{i_x}$ from $\mathbf{S}_x$ and $\mathbf{s}_{\mathbf{i_y}}$ from $\mathbf{S_y}$ and calculating the dot product $\mathbf{s}_{i_x} \cdot \mathbf{s}_{i_y}$ of the setvectors in order to obtain a single value. A low valued dotproduct implies a good match of the applications, a high dotproduct value suggests a bad match, i.e. a high level of cache interference resulting in many cache misses.

The dotproducts do not have any specific meaning like *number of additional cache misses*, as it is the case with Chandra's circular sequence based method. However, comparing the dotproducts of several thread combinations *in relation to each other* has been proven to be an effective way to predict which threads make a better match and which threads do not.

4 Evaluating Cache Contention Prediction Techniques – Simulation Setup

In order to prove the effectiveness of the setvector method with its relative comparison of dotproducts, we compared it to Settle's activity vector method and to Chandra's circular sequence based method. We refrained from additionally comparing the setvector method to Chandra's stack distance based method, as Chandra already reported that the circular sequence based method outperformed the stack distance based methods – and our setvector method showed nearly the same accuracy as the circular sequence based method.

To compare and evaluate the cache contention prediction techniques, we generated tracefiles with memory accesses representing 512 million instructions for each of the ten SPEC2006 benchmark programs *astar*, *bzip2*, *gcc*, *gobmk*, *h264ref*, *hmmer*, *lbm*, *mcf*, *milc* and *povray* applying the *Pin* toolkit [4]. Of these ten programs, we executed every 45 pairwise combinations on our MCCCSim multicore cache contention simulator [8] that had been parameterized as follows:

Parameter	private L1 cache	shared L2 cache
Size	32 k	2 MB
Line size	128 Byte	128 Byte
Associativity	2	8
Hit time	1.0 ns	10.0 ns
Miss time	depends on L2	100.0 ns
Replacement	LRU	LRU

For each program of each combination, we calculated the difference between the stand-alone memory access time and the memory access time when executed in co-schedule with the other application. From this difference, we calculated the additional penalty in picoseconds per instruction, that is shown for every combination in table 3a). Additionally, we sorted the results according to 1st) this penalty and 2nd) the application's name.

Then, we calculated the predictions for the *activity vector* method, Chandra's *circular sequence* based method and our *setvector* method and sorted them accordingly, as can be seen from table 3b) - 3d).

To evaluate the prediction methods, we introduce a method we call *mean ranking difference* (MRD): We compare the rows of table 3a) that represent values gathered from MCCCSim with those of the predictions, exemplarily shown in table 3b) - 3d). Figure 4 shows that we calculate the absolute difference between the position (ranking) determined by MCCCSim and the position determined by the prediction for each combination. The results are summed up and divided by the total number of co-scheduled applications (9) to yield the average mean ranking distance (MRD), i.e. the mean number of positions, a co-schedule's prediction differs from the real values obtained from MCCCSim.

We evaluated several variations of all three methods.

With Chandra et al.'s method, we were interested in comparing the predictions for the following variations:

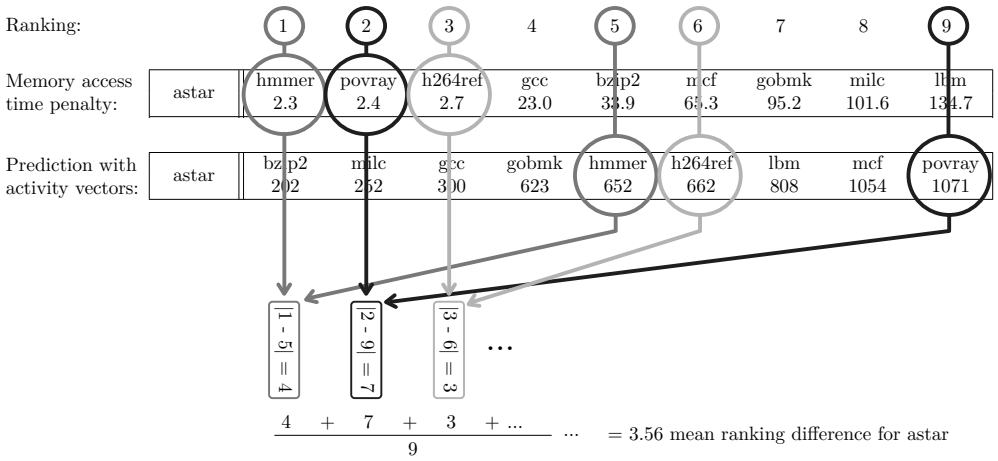

Figure 4: Determination of the mean ranking difference (MRD) for *astar*.

- *Chandra cseq chunkset*: Prediction while applying only one circular sequence stack to a chunkset (i.e. interval of 2^{20} instructions).
- *Chandra cseq Af(set)*: Prediction while applying a circular sequence stack to every cache set within an interval and measuring the memory access frequency on a *per cache set basis*.
- *Chandra cseq Af(chunkset)*: Prediction while applying a circular sequence stack to every cache set within an interval without partitioning the memory access frequency on the cache sets, i.e. providing only *one* memory access frequency value per interval.

Settle et al. stated that "the low order bits of the cache set component of a memory address are used to index the activity counter associated with each cache super set." [6] However, we expected that the method would achieve better results when using the *high* order bits to index the activity counters since addresses with equal high order bits are mapped to equal cache sets. Therefore, we evaluated the activity vector method for these two variants naming them *high* respectively *low* (cf. table 1).

With the setvector method, we were interested in analyzing the following variations (cf. table 1):

- *diff. x access*: The setvector method as it had been presented in chapter 3.
- *access*: Utilizing only the access frequency. This way, the performance of the activity vector method can be estimated for the case that the number of supersets reaches its maximum (i.e. the over all number of sets) and the activity expresses the number of accesses to a set and not just the one-bit information, whether or not a specific threshold has been reached.
- *diff*: Utilizing only the number of different cache lines that are mapped to the same cache set, i.e. ignoring any access frequency.
- *add, mul*: Combining the vectors of two threads by applying either elementwise addition or multiplication and calculating the average of the elements afterwards, rather than by applying the dot product.

5 Results

Table 1 shows the accuracy of the evaluated methods and variations, table 2 shows the execution time of the methods, subdivided into time that has to be spend *offline* (row *cseq profiling* and *vector creation*), and the time that has to be spend *online* (row *prediction*) when calculating the prediction for a specific combination. Table 1 shows that Chandra's circular sequence based method that utilizes the access frequency on a *per set* basis performs with the highest accuracy ($MRD = 0.58$). However, 676.83 picoseconds have to be spent per instruction (ps/instr.) on average to calculate the predictions, i.e. prediction takes about 6768 times longer than for the activity vector method (0.10 ps/instr.) and about 3981 times longer as for the setvector method.

Although the activity vector method performs quite fast, it shows a high error rate ($MRD = 3.07$ and $MRD = 2.38$ respectively). However, selecting the higher part of the set bits had been a good idea. Increasing the number of super sets to the number of sets and applying natural numbers to count the number of accesses to each set instead of using only a single bit per set significantly improves accuracy ($MRD = 0.64$, as seen from *Setvector – access, add*), but also increases prediction time (0.16).

The setvector method that utilizes both access frequency and number of accesses from different keys shows about the same prediction time (0.17 ps/instr.), but a slightly better accuracy ($MRD = 0.60$), that nearly matches that of the about 3981 times slower circular sequence based method.

	Chandra cseq chunkset	Chandra cseq Af(set)	Chandra cseq Af(chunkset)	Activityvector low	Activityvector high
astar	1.56	0.89	0.89	3.56	2.00
bzip2	0.89	0.44	0.89	2.67	1.33
gcc	0.89	0.67	0.89	3.11	2.00
gobmk	0.67	0.67	0.44	3.11	3.33
h264ref	0.67	0.67	0.89	2.67	2.44
hmmer	0.89	0.67	1.11	2.89	2.44
lbm	1.11	0.67	1.33	4.00	2.22
mcf	0.44	0.22	1.33	3.11	2.22
milc	0.67	0.00	0.44	2.89	3.11
povray	2.00	0.89	0.89	2.67	2.67
average	0.98	0.58	0.91	3.07	2.38

	Setvector diff. x access	Setvector access, add	Setvector access, mul	Setvector diff., add	Setvector diff., mul
astar	0.67	0.67	0.44	0.89	0.89
bzip2	0.67	0.67	0.22	0.44	0.89
gcc	0.89	0.89	0.67	0.67	0.67
gobmk	1.11	1.33	1.56	0.89	0.67
h264ref	0.44	0.44	0.44	1.11	1.11
hmmer	0.22	0.22	0.67	0.89	0.89
lbm	1.33	1.33	1.56	0.89	0.44
mcf	0.00	0.00	0.89	0.22	0.22
milc	0.22	0.44	0.89	0.22	0.44
povray	0.44	0.44	0.22	1.11	1.11
average	0.60	0.64	0.76	0.73	0.73

Table 1: Mean ranking difference (MRD) for each benchmark and method.

6 Conclusion

In this paper, we presented state-of-the art methods to predict cache contention and proposed a new prediction method based on the calculation of so-called *setvectors*. We simulated the additional memory access time introduced by cache contention during application coscheduling and compared those values to the prediction methods by applying a new metric called *MRD* (mean ranking distance) that calculates the mean difference between the predicted and the simulated ranking.

Our results showed that the method introduced by Chandra et al. [1] might be the most accurate one, but it is nearly 4000 times slower than the proposed setvector method, that achieves nearly the same accuracy ($MRD = 0.60$ instead of $MRD = 0.58$).

References

[1] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*, 2005.

[2] A. Fedorova. *Operating System Scheduling for Chip Multithreaded Processors.* PhD thesis, Harvard University, Cambridge, Massachusetts, 2006.

[3] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. In *IEEE Transactions on Computers*, volume 38, 1989.

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klausner, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation, ACM*, 2005.

[5] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, volume 9, 1970.

[6] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural Support for Enhanced SMT Job Scheduling. *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques*, September 2004.

[7] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1), March 1995.

[8] M. Zwick, M. Durkovic, F. Obermeier, W. Bamberger, and K. Diepold. MCCCSim - A Highly Configurable Multi Core Cache Contention Simulator. *Technical Report - Technische Universität München, https://mediatum2.ub.tum.de/doc/802638/802638.pdf*, 2009.

[9] M. Zwick, M. Durkovic, F. Obermeier, and K. Diepold. Setvectors for Memory Phase Classification. In *International Conference on Computer Science and its Applications (ICCSA'09)*, 2009.

Table 2: Comparison of the execution times of the prediction methods.

Method		nanoseconds per instruction for task										average	sum
		astar	bzip2	gcc	gobmk	h264ref	hmmer	lbm	mcf	milc	povray		
Chandra cseq; prediction with access frequency per set	cseq profiling	410.28	393.03	417.72	401.39	405.74	387.03	436.39	401.13	430.92	409.72	409.34	1086.17
	prediction	680.07	728.65	648.39	716.01	541.98	603.82	810.44	699.79	789.42	549.75	676.83	
Chandra cseq; prediction with access frequency per chunkset	cseq profiling	410.28	393.03	417.72	401.39	405.74	387.03	436.39	401.13	430.92	409.72	409.34	1155.39
	prediction	679.09	724.70	649.47	848.25	736.89	595.34	802.37	708.18	980.43	735.81	746.05	
Chandra cseq; prediction per chunkset	cseq profiling	410.28	393.03	417.72	401.39	405.74	387.03	436.39	401.13	430.92	409.72	409.34	666.64
	prediction	297.21	267.22	263.03	263.97	264.33	261.29	262.71	260.32	259.48	173.50	257.31	
Activityvector, low	vector creation	52.34	48.15	42.42	45.21	46.14	48.60	60.69	41.75	54.32	48.24	48.79	48.88
	prediction	0.09	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	
Activityvector, high	vector creation	52.34	48.15	42.42	45.21	46.14	48.60	60.69	41.75	54.32	48.24	48.79	48.88
	prediction	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	
Setvector; diff. x access	vector creation	55.51	52.69	44.05	50.76	48.18	51.69	83.08	46.79	61.35	51.32	54.54	54.71
	prediction	0.16	0.17	0.16	0.17	0.16	0.16	0.18	0.17	0.17	0.17	0.17	
Setvector; access, add	vector creation	55.51	52.69	44.05	50.76	48.18	51.69	83.08	46.79	61.35	51.32	54.54	54.71
	prediction	0.16	0.16	0.16	0.17	0.16	0.16	0.17	0.16	0.16	0.16	0.16	
Setvector; access, mul	vector creation	55.51	52.69	44.05	50.76	48.18	51.69	83.08	46.79	61.35	51.32	54.54	54.71
	prediction	0.16	0.16	0.16	0.17	0.16	0.16	0.17	0.16	0.17	0.17	0.16	
Setvector; diff., add	vector creation	55.51	52.69	44.05	50.76	48.18	51.69	83.08	46.79	61.35	51.32	54.54	54.71
	prediction	0.16	0.16	0.16	0.17	0.16	0.16	0.17	0.17	0.17	0.16	0.16	
Setvector; diff., mul	vector creation	55.51	52.69	44.05	50.76	48.18	51.69	83.08	46.79	61.35	51.32	54.54	54.71
	prediction	0.16	0.17	0.16	0.17	0.16	0.16	0.17	0.17	0.17	0.16	0.17	

a)

1st task	co-scheduled 2nd task Penalty in picoseconds per instruction for 1st task as simulated by MCCCSim								
astar	hmmer 2.3	povray 2.4	h264ref 2.7	gcc 23.0	bzip2 33.9	mcf 65.3	gobmk 95.2	milc 101.6	lbm 134.7
bzip2	h264ref 9.5	hmmer 10.2	povray 15.2	astar 37.3	gcc 84.5	mcf 162.4	gobmk 165.4	milc 193.3	lbm 311.3
gcc	hmmer 2.3	h264ref 5.2	povray 5.7	astar 28.1	bzip2 88.1	mcf 113.9	gobmk 119.6	milc 128.8	lbm 196.5
gobmk	hmmer 0.8	h264ref 1.1	povray 1.4	astar 3.0	gcc 4.6	bzip2 7.2	milc 11.7	mcf 12.1	lbm 21.0
h264ref	hmmer 0.0	povray 0.6	astar 1.2	gcc 2.4	bzip2 4.0	mcf 10.5	milc 10.7	gobmk 19.0	lbm 20.4
hmmer	h264ref 0.0	povray 0.1	gcc 0.3	astar 0.3	bzip2 1.9	milc 10.1	mcf 13.0	gobmk 21.0	lbm 48.0
lbm	h264ref 0.0	hmmer 0.0	povray 0.0	gcc 0.0	astar 0.0	mcf 0.0	bzip2 0.0	milc 0.0	gobmk 0.1
mcf	h264ref 13.1	povray 16.6	hmmer 17.8	astar 45.5	gcc 58.8	bzip2 98.6	gobmk 160.0	milc 194.8	lbm 275.0
milc	h264ref 31.1	povray 36.3	hmmer 45.2	astar 148.3	gcc 151.5	bzip2 270.1	mcf 387.4	gobmk 403.0	lbm 570.7
povray	hmmer 0.2	h264ref 0.3	astar 1.2	gcc 3.2	bzip2 8.3	mcf 10.0	milc 11.0	gobmk 13.5	lbm 27.6

b)

1st task	co-scheduled 2nd task Prediction with the setvector method, diff x access – cache set granularity								
astar	povray 194	hmmer 672	h264ref 800	gcc 9676	bzip2 20104	milc 162787	mcf 179980	gobmk 299971	lbm 1896607
bzip2	povray 9227	h264ref 9544	hmmer 10764	astar 20104	gcc 43291	mcf 298288	milc 323193	gobmk 366661	lbm 2227396
gcc	povray 53	h264ref 293	hmmer 2465	astar 9676	bzip2 43291	milc 185386	mcf 223167	gobmk 302504	lbm 2068453
gobmk	povray 274080	h264ref 281287	astar 299971	gcc 302504	hmmer 302958	bzip2 366661	mcf 549299	milc 675690	lbm 2369176
h264ref	hmmer 0	povray 8	gcc 293	astar 800	bzip2 9544	milc 89448	mcf 126345	gobmk 281287	lbm 1839296
hmmer	h264ref 0	povray 0	astar 672	gcc 2465	bzip2 10764	milc 112992	mcf 153425	gobmk 302958	lbm 1879465
lbm	povray 1833244	h264ref 1839296	hmmer 1879465	astar 1896607	gcc 2068453	bzip2 2227396	gobmk 2369176	mcf 2713675	milc 2864029
mcf	h264ref 126345	povray 130910	hmmer 153425	astar 179980	gcc 223167	bzip2 298288	gobmk 549299	milc 709719	lbm 2713675
milc	h264ref 89448	povray 90127	hmmer 112992	astar 162787	gcc 185386	bzip2 323193	gobmk 675690	mcf 709719	lbm 2864029
povray	hmmer 0	h264ref 8	gcc 53	astar 194	bzip2 9227	milc 90127	mcf 130910	gobmk 274080	lbm 1833244

c)

1st task	co-scheduled 2nd task Prediction with activityvectors – 32 super sets								
astar	bzip2 202	milc 252	gcc 300	gobmk 623	hmmer 652	h264ref 662	lbm 808	mcf 1054	povray 1071
bzip2	astar 202	hmmer 561	gcc 588	mcf 644	lbm 695	gobmk 878	h264ref 926	milc 1097	povray 1176
gcc	astar 300	mcf 396	hmmer 572	bzip2 588	milc 860	lbm 973	h264ref 1170	gobmk 1238	povray 1660
gobmk	astar 623	bzip2 878	hmmer 1175	gcc 1238	mcf 1360	milc 1383	lbm 1509	h264ref 1804	povray 2417
h264ref	astar 662	bzip2 926	mcf 1079	gcc 1170	milc 1270	hmmer 1488	lbm 1593	gobmk 1804	povray 2478
hmmer	milc 343	bzip2 561	gcc 572	mcf 611	astar 652	gobmk 1175	povray 1341	h264ref 1488	lbm 1536
lbm	bzip2 695	astar 808	milc 891	gcc 973	mcf 1148	gobmk 1509	hmmer 1536	h264ref 1593	povray 1646
mcf	gcc 396	hmmer 611	bzip2 644	milc 936	astar 1054	h264ref 1079	lbm 1148	gobmk 1360	povray 1768
milc	astar 252	hmmer 343	gcc 860	lbm 891	mcf 936	bzip2 1097	h264ref 1270	gobmk 1383	povray 1562
povray	astar 1071	bzip2 1176	hmmer 1341	milc 1562	lbm 1646	gcc 1660	mcf 1768	gobmk 2417	h264ref 2478

d)

1st task	co-scheduled 2nd task Additional misses predicted by the cseq method – calculation per cache set								
astar	povray 2648	h264ref 3263	hmmer 4509	gcc 32309	mcf 41069	bzip2 41741	milc 73348	gobmk 122598	lbm 126190
bzip2	h264ref 17021	hmmer 22684	povray 28402	astar 66146	gcc 173455	milc 323414	mcf 478706	gobmk 547305	lbm 819350
gcc	h264ref 3718	hmmer 4623	povray 4633	astar 33950	bzip2 109747	milc 135006	mcf 209894	gobmk 230418	lbm 362239
gobmk	h264ref 898	povray 951	hmmer 1357	gcc 3103	astar 3644	bzip2 7445	milc 9957	mcf 17750	lbm 28811
h264ref	hmmer 19	povray 202	gcc 238	astar 247	milc 2411	bzip2 3824	mcf 7874	gobmk 14457	lbm 19004
hmmer	h264ref 27	gcc 30	povray 32	astar 54	milc 489	mcf 533	bzip2 1003	gobmk 1502	lbm 2524
lbm	h264ref 0	hmmer 0	povray 2	astar 12	gcc 13	bzip2 247	milc 934	mcf 17740	gobmk 33327
mcf	h264ref 34600	povray 38415	hmmer 44746	astar 99232	gcc 168854	bzip2 302367	milc 568534	gobmk 668179	lbm 1178915
milc	h264ref 83561	povray 92462	hmmer 130951	astar 313825	gcc 327676	bzip2 532395	mcf 1009692	gobmk 1034280	lbm 1199336
povray	h264ref 316	astar 379	hmmer 445	gcc 2797	milc 3404	mcf 6893	bzip2 10277	gobmk 12406	lbm 29227

Table 3: Co-scheduling penalty and its prediction.