# Virtually Splitting Data Caches using Multiple Address Decoders

Tomislav Janjusic and Krishna Kavi

University of North Texas, Denton TX 76209, USA,
tjanjusic@unt.edu,
WWW home page: http://csrl.unt.edu

**Abstract.** An address decoder is a small hardware unit that uses an address to index and place the data into memory units including cache memories. In current CPU cache designs there is a single decoder unit which serves to place data into the cache. In this paper we describe a technique to reduce contention on CPU's caches through the use of multiple address decoders. We argue that by using multiple decoding techniques better data placement can be achieved and the CPU cache can be better utilized. We present an overview of an instrumentation tool developed to collect fine-grained data traces and a technique for virtually splitting caches using separate address decoders. Our results demonstrate the feasibility and the impact of virtual cache splitting.

**Keywords:** data caches, instrumentation, cache splitting

## 1 Introduction

The CPU-Memory speed gap persists to this day while cache designs remain virtually unchanged. The processor-memory speed gap refers to the ever increasing speed of processors when memory access speeds are lagging. This results in the CPU sitting idle waiting for requested data to be brought from main memory. To reduce the speed gap computer architects introduced CPU caches.

A CPU's cache is a fast but space-limited storage unit that serves as an intermediary place holder for data that is requested by the CPU. The speed difference between main memory and a CPU's cache is primarily due the design and due to their distance to the CPU.

The introduction of caches diminished the negative effects of the speed-gap, but it was soon realized that to further improve cache effectiveness it is worthwhile to explore different placement and replacement policies. This led computer architects to introduce multi-level caches, higher associative caches, better replacement techniques, etc.

As software became more complex and the average applications' memory footprint increased, a CPU's cache exhibited greater cache thrashing due to the increased data contention per cache line. Several studies explored various addressing schemes to improve this behavior [1–5], but none have addressed the issue from a data-type perspective.

## 1.1 Cache Placement

It is important to understand the components of a computer system which drive cache data placement because they are the driving factor of cache related performance issues. At the top level we have page placement driven by the kernel. The physical pages assigned by the kernel relate to physical memory which maps *physical addresses* to the application's virtual space. In modern systems a shared and last level cache (LLC) is indexed using a data's physical address. This allows the system to reduce data and instruction duplication across multiple cores and reduces address translation cost.

Processes see only their own virtual space. Private level caches (e.g., L1 caches) are indexed using the virtual address. Depending on the data layout this becomes a factor behind performance issues related to private level caches. The virtual space data layout is controlled by three agents: the compiler, the user, and the run-time system's memory allocator. The compiler reorders static data usually placed within the stack and the uninitialized sections. The user exhibits control over structure data placement. The run-time system controls dynamic data allocation and generally impacts fragmentation and an application's memory footprint. These factors determine data's virtual address placement.

At the lowest level we have the cache memory controller which controls where the data is to reside once requested by the CPU. The design is fairly simple, depending on the cache level, an address decoder's input is either a virtual or physical address. The decoder translates the address to determine the cache line to index. Depending on the cache size, line size, and a cache's associativity, fixed number of bits are used to index the cache. It is important to note that for all data-types the same addressing scheme is used because in todays processors only a single decoder is used.

$$\boxed{\overbrace{TAG|\underbrace{INDEX}_{Log_{(2)}(Sets)}|\underbrace{OFFSET}_{mbits}}^{address}}$$

Fig. 1. A data's address and its corresponding bits used to index into the cache.

Figure 1 shows how a decoder uses bits of an address to search for data in the cache. For example assume an address space of $2^N$ bytes and a cache size of $2^{n+b}$ bytes (with $2^n$ lines and $2^b$ bytes per line). An address decoder uses $m$ bits out of the N address bits to locate a set with k lines (k-way associative), where $m = n - log_2(k)$, and we use $b$ additional bits to locate a byte. This means that $N - m - b$ bits are used as the tag.

## 1.2 Trace collection and simulation

In order to observe how an application utilizes a CPU's cache the traditional miss-rate statistics are not enough. A miss-rate statistic is a good measure if we wish to obtain a general picture of an application's impact on the cache; however, to determine cache utilization we must collect statistics for each cache set. To measure the effects of different data-types on cache utilization we must collect enough information about each cache access of an application and this information can be used to evaluate newer cache designs that may improve cache performance based on accesses to specific data objects. This is the focus of our research and the focus of this paper.

Therefore make three key contributions:

1. We describe a tool that allows us to track every memory access and relate it back to source level object names. This information can be used to analyze functions, and data structures that cause most memory performance bottlenecks.
2. We describe a method for separating objects into different cache partitions to minimize conflicts. More specifically we show the performance gains achieved by separating stack, heap, and global areas of applications into separate cache partitions.
3. We show how to adjust the size of cache partitions allocated for each program data segment and how proper selections of these sizes is important to achieve improved performance.

The rest of the document is organized as follows; in Section 2 we will review related work. There has been much work done in the area of improving cache utilization, reducing cache miss rates, and cache designs to improve overall system performance. In section 3 we will provide a basic overview of the tools that we use and our experimental methodology. In order to obtain detailed information we rely on trace-driven cache simulation. As part of our research we have developed a set of tools to facilitate this research. In section 4 we present our results. In section 5 we will offer our conclusions, and finally in section 6 we will discus future research direction.

## 2 Related Work

Improving cache placement has been discussed extensively from both the software and hardware perspective. Compiler research and memory allocation techniques address the software aspect of optimal data placement [6–9]. Hardware research concentrates on reducing cache conflicts by means of applying various cache indexing schemes or introducing additional hardware. The ultimate goal of either research is performance optimization.

Over the past decade several researchers reported on using various indexing schemes to reduce overall cache conflicts. In [10] several of these techniques have been evaluated in terms of their overall applicability. The conclusion is that cache indexing scheme are application dependent and thus their performance gains are variable.

From a cache's perspective achieving optimal placement generally falls within two categories: *optimal indexes* and *dynamic cache remapping.*

## 2.1 Optimal Indexing

Assume an application's trace is known at runtime. An optimal index consists of selecting best possible bits as index bits (see Figure 1) that map the address to a cache set. In a more general way this can be viewed as a hashing function. Therefore the goal is to find bits which define a perfect hashing function. However, there are two shortcoming with this approach. Firstly for the majority of these approaches the trace must be known and secondly finding a perfect hashing function is NP-complete [10].

## 2.2 Dynamic Cache Remapping

The other approach is by introducing additional hardware units. Higher associative caches lead to better cache utilization and reduction of cache misses. Increasing a cache's associativity allows multiple data elements to map to the same index. This also means that replacement functions must be introduced that lead to greater complexity, increased access delays and cost. We must note that higher associativity does not predominate in higher level caches (i.e., caches closer to the CPU). Even in modern processors higher level (L1) caches exhibit 2-way or 4-way associativity. Therefore researchers proposed dynamic relocation of addresses through additional hardware units.

**Column Associativity** In [1], a technique known as column associative caches is proposed to reduce misses in directly mapped (DM) caches. In this technique a cache is viewed as DM and if the element is not found then the cache is viewed as 2-way associative. This provides higher associativity only when needed.

**Adaptive Caches** In this approach conflicting data items are relocated to new cache lines based on two history tables [5]. The two tables keep a list of most recently used set references *set reference history table (SHT)*, and an OUT table that maintains indexes for items evicted from heavily referenced sets. The idea is to relocate items from heavily accessed sets (maintained in SHT) to new sets and the new set indexes are kept in OUT. The performance of this technique depends on the size of the two tables.

## 2.3 Cache Splitting

In our previous work we discussed the feasibility of splitting the data cache into array specific and scalar specific data caches [11]. This technique should not be confused with todays instruction and data split caches which are common in modern processors. Splitting data caches into array and scalar caches reduces cache conflicts commonly occurring between high spatial locality structures such as arrays and low spatial locality structures such as dynamic elements [11].

There are several observable correlations with a poorly utilized cache. Because cache performance relates to cache utilization, an underutilized cache implies poor performance. It was reported in [12] that cache accesses remain highly non-uniform even with increased associativity. High non-uniformity implies that majority of data items are mapped to a small number of cache sets. This observation has a couple of implications: power is wasted due to unused cache lines, and mapping of multiple data items to a subset of cache implies higher contentions to those sets, leading to higher miss rates even when other sets are not used.

**Virtual Cache Splitting** In this paper we present a technique to virtually split caches using multiple address decoders. As previously explained it is possible to vary data mapping based on an indexing scheme. We make several assumptions regarding the feasibility of this approach. We assume that the architecture provides new *load* and *store* address instructions. These instruction simply indicate which decoder to use. The use of these instructions also assumes that the compiler technology can differentiate different object types and use different decoders. For this paper, we identify and use different decoders for stack, heap, globals, etc. These assumptions are not far fetched, because manipulating stack frame pointers and managing heap sections is a fundamental task of any compiler.

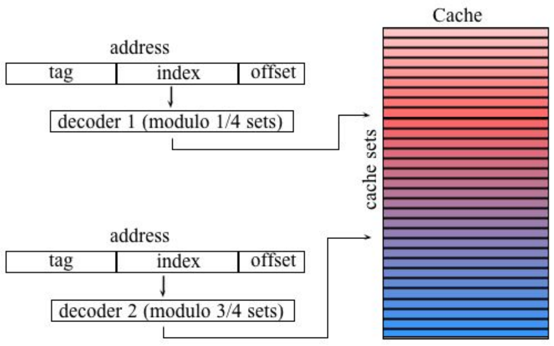

Fig. 2. Multiple Address Decoders

Figure 2 depicts the use of multiple address decoders. It is for illustration purposes only because to address $\frac{3}{4}$ of a cache we can use 3 decoders each addressing $\frac{1}{4}$, thus minimizing the hardware needed by the decoders.

## 3 Simulation Methodology

### 3.1 Introduction

Our simulation incorporates several tools developed for the specific purpose of this research. We rely on a binary instrumentation tool called Gleipnir [13] to provide the detailed trace information required for our study. Our cache simulator is based on DineroIV [14] a configurable trace driven simulator. Our modification are small in that the simulator takes into account additional information provided by Gleipnir.

## 3.2 Gleipnir

Gleipnir is built on top of Valgrind [15], a binary instrumentation framework. It is similar to the standard Valgrind tools such as *Lackey*, a basic instruction tracing tool and *Massif* a heap profiler.

Gleipnir operates on a set of events provided by Valgrind. Each event is either an Ir, Dr, Dw, or Dm for Instruction read, Data read, Data write, or Data modify respectively. Every instruction is recorded and traced. During the execution of data read, write, or modify, the address is fed into a debug parser which will search the known debug symbol table and output corresponding information regarding source level program elements back into the trace. For static and global variables this will suffice, however, for dynamically allocated objects Gleipnir uses a wrapper to Valgrind's allocation routines to intercept calls and record the blocks. Symbol table look-up enables Gleipnir to deliver fine-grained information for each data write, read, or modify. Every instruction is annotated with the address to be fetched, modified, or written to; the function which caused the access; the scope of the variable, thread that executed the code; and finally the data element itself. See Table 1 for the format of traces generated by Gleipnir.

| ACCESS TYPE | ADDRESS | FUNCTION | SCOPE | FRAME NO. | THREAD | VARIABLE |
|---|---|---|---|---|---|---|
| S | 7ff000108 | malloc | LS | 0 | 1 | _zzq_args[5] |

Table 1. Gleipnir's trace line

The individual elements are easily accounted for if the variable is a global or local (stack) data-structure or variable. The first field is the access type, either a *Load, Store, Modify, Instruction, or X (for miscellaneous instructions)*. The second field is the *virtual address* of the data to be accessed followed by the function name. If any symbol information exists the trace will be annotated with the element's scope (*Local*,*Global*, or *Heap*), and the element's type ( *Variable* or *Structure*). The next two numbers indicate the elements *Frame* and the executing *Thread*. The final value is the variable name itself.

Because Gleipnir relies on Valgrind's internal debug parser to fetch the information applications must be compiled with the compiler's *-g* flag.

## 3.3 Simulation

Our approach uses a modified cache simulator based on DineroIV [14]. DineroIV is a trace driven uniprocessor cache simulator. Our extensions are based on Gleipnir's trace information. This allows our cache simulation to track cache access statistics for each program data's segment and all identified program structures. Simulator's output consists of cache set statistics for each function and encountered variables. The simulations results are used for further analyses. Table 2 shows a function cache miss summary after a Mibench *patricia* [16] benchmark simulation. The output is used to identify functions with most cache misses.

| misses | function name | variables |
|---|---|---:|
| 105221 | _HEAP_ | 0 |
| 60570 | pat_search | 842 |
| 44613 | _IO_vfscanf | 4 |
| 43891 | main | 32686 |
| 28702 | __printf_fp | 1 |
| 21910 | str_to_mpn | 1 |
| 21780 | ____strtol_l_internal | 1 |
| 14327 | xsputn@@GLIBC_2.2.5 | 1 |
| 10996 | rawmemchr | 2 |
| 9624 | memchr | 1 |
| 4986 | vfprintf | 1 |
| 2211 | unided | 1 |
| 1114 | ____strtof_l_internal | 1 |
| 917 | puts | 1 |
| 666 | printf | 1 |
| 630 | _itoa_word | 1 |
| 622 | memcpy | 2 |
| 553 | fgets | 3 |
| 504 | _IO_getline_info | 3 |
| 484 | __mpn_mul | 1 |
| 371 | insertR | 456 |
| 368 | _STACK_ | 0 |

Table 2. Simulation results after using separate address decoders.

Because Gleipnir's traces provide meta-data information that describe accesses based on their scope we can explore the performance impact of reserving special areas in the cache for data within the same scope. It is now possible to group all stack based accesses and group all dynamic accesses into reserved cache areas. We achieve this by tracking an access scope. An instruction which is identified as a local structure or variable (*LS or LV*) will use a stack decoder. A stack decoder is different from the standard decoder in that its modulo arithmetic will only include a fraction of total cache sets. This will force all stack based accesses to be grouped into a reserved cache area. Similarly any heap access (*HB*) will use a heap decoder that groups dynamic accesses into a different portion of the cache. Global (*GV or GS*) accesses use a global decoder. The amount of available scope information is application dependent. In addition we can track the application expanding and retreating stack's address, global address, and dynamic region to account for 100% of all *stack, heap, and global* accesses.

## 4 Results

Our intent is to simulate an environment in which compilers or runtime systems can choose decoders based on application's specific needs. For example, an application which predominately uses *heap* elements will require a larger portion of the cache reserved for dynamic accesses. Likewise an application that uses *stack* data will require larger portion of the cache reserved for its stack. This applies to *global* data as well.

We claim two hypotheses: Our first hypothesis is that we can improve cache performance by splitting the CPU cache into regions reserved for specific data segments. Our second hypothesis is that cache region size is application dependent, and they should be carefully selected to optimize cache performance. We use a variety of MiBench benchmarks[16] to validate our hypotheses.

Using a single decoder is the standard decoding scheme in which we use a single decoder to place data into the cache. To our knowledge all of today's architectures use a single decoding scheme which makes no distinctions of fetched data types. Mibench benchmarks are applications with relatively small memory footprints; therefore, we have reduced the cache size to 32k bytes and 32 bytes per block line.

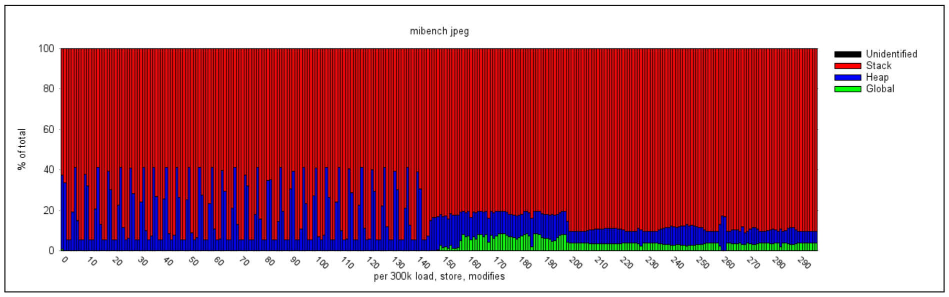

Fig. 3. Jpeg's *stack, heap, and global* references

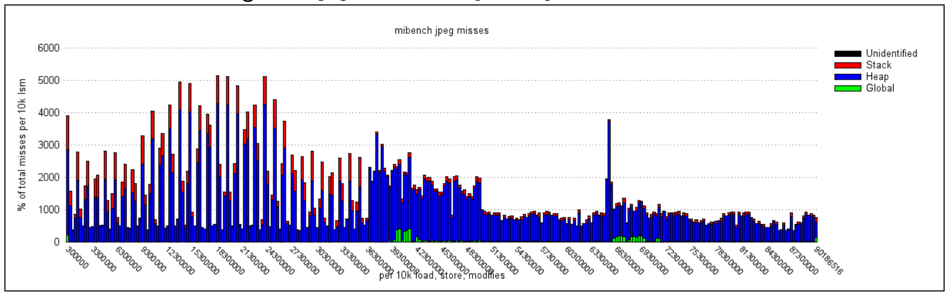

Fig. 4. Jpeg's *stack, heap, and global* misses

Figure 3 depicts reference counts for the benchmark *jpeg*. The $X$ axis are the number of references during program execution. The $Y$ axis is the percent of total references for *stack, heap, and global* data. It can be observed that the majority of accesses are stack references (red). They consume on average over 80% of all references. Figure 4 shows the number of misses for each segment per

10k misses. It can be observed that in the first 40% of all misses the stack misses exhibit a greater miss count than in the later 60%. Interestingly our results in Figure 4 show that majority of misses occur between heap data. This implies that references to stack data are evicting heap data.

Figure 3 and Figure 4 also show that an application may exhibit distinct phases of segment utilization.

After running several mibench benchmarks we observed that separating data segments into reserved cache areas benefits the overall cache behavior.

Figure 5 shows the overall miss reduction for a variety of mibench benchmarks. In all but two benchmarks a split cache design improves overall behavior. The two benchmarks that do not benefit from a split cache design are *basicmath* and *bitcount*. The reason is that both benchmarks operate strictly from the stack segment. This means that any reduction in cache space for the stack will have diminishing overall cache effects.

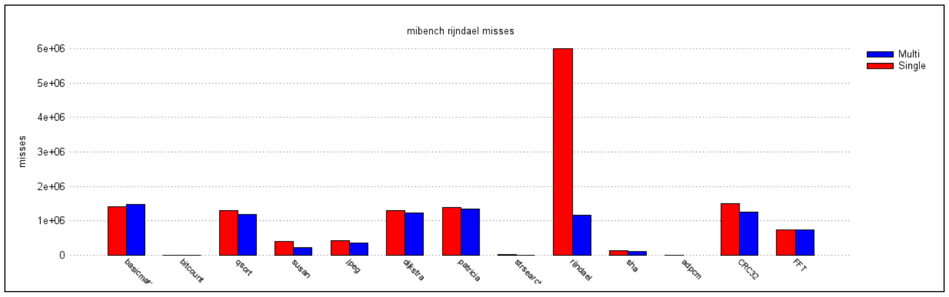

**Fig. 5.** Jpeg's *stack, heap, and global* misses

Figure 6 shows normalized cache effects. It can be observed that on average a split cache design shows a ≈25% improvement. The outliers are *basicmath* and *bitcount* benchmarks.

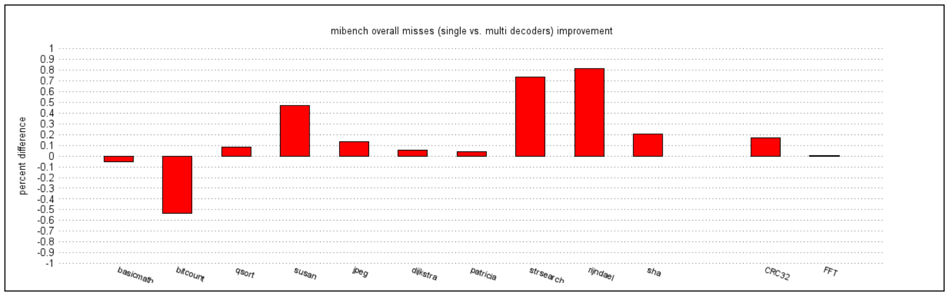

**Fig. 6.** Jpeg's *stack, heap, and global* misses

### 4.1 Design Cost

We make two assumptions regarding our cache design. First, we require that there exists architectural support which allows the compiler or runtime system to choose which decoder to use. Additional decoder cost are minimal because a decoder requires simple logic to build. In most cases a register and multiplexer will suffice to build a decoding unit.

The second assumption is that required compiler support exists. Managing an application's stack top address and the global address segment is a fundamental task of any compiler; therefore we feel that that these assumptions are not far fetched in terms of cost or future system implementation.

## 5 Conclusions

In this paper we started with two hypotheses. Our first hypothesis claimed that we can reduce cache misses by reserving areas in data cache (or splitting cache) for different data segments. Our second hypothesis claimed that because applications behave differently and different data segments vary in size for different application, no single cache splitting technique will satisfy all applications. In this paper we have described a technique to decouple data access patterns into *stack, heap, and global* accesses. Our experiments show the potential to improve cache performance by decoupling accesses based on data regions. We used trace driven cache simulation experiments to simulate data caches split into different regions, each portion reserved for a specific data access region (e.g., stack, heap and globals). Using these experiments we validated these hypotheses.

In this research we relied on two tools: Gleipnir [13] and DineroIV [14]. Gleipnir was developed by us to collect fine-grained trace information on memory accesses and relate them back to source level objects. We used the fine grained information for identifying accesses to different data segments like stack, heap and globals. It should be noted that, to our knowledge, no other tool or set of tools that can provide the detailed information on memory access as Gleipnir. Gleipnir can be used for other purposes beyond that described in this paper.

DineroIV is a trace driven cache simulator which was modified for our purpose so that we could split data caches into different regions. We outlined how this can be achieved using multiple address decoders, one per region. While this requires a very modest additional hardware, our experiments show that the performance gains resulting from the elimination of cache conflicts outweigh the additional hardware investment.

## 6 Future Work

### 6.1 Structure based cache splitting

Gleipnir's traces provide meta-data information about local variables and local structures. They are identified with either a $V$ for variable or $S$ for structure.

A structure may be an array or a dynamically allocated *struct* type. Knowing this information we should be able to split data caches by reserving sections for specific data types. In our future experiments we will explore the feasibility of reserving structure specific cache regions.

### 6.2 Dynamic Cache Partitioning

Our experiments rely on static cache configuration. This means that our simulation must decide beforehand which type of partitioning to use. Our cache partitions are set based on previous simulation analyses of an application's *stack, heap, or global* segment behavior. While for many practical purposes this may not be ideal it still shows the feasibility of segment based cache partitioning. We are actively exploring an effective and cost-friendly design to dynamically adjust our cache partition sizes depending on application's run-time behavior.

### 6.3 Multithreaded Cache Simulations

Valgrind's framework enables Gleipnir to collect trace information for multi-threaded processes. However, Valgrind segregates memory accesses on a per thread basis. No information about the order of accesses of multiple threads is maintained. We are currently exploring models for interleaving thread accesses in order to create a simulation of conflicts among multiple threads. Another related problem is Gleipnir's use of virtual addresses instead of physical addresses. Use of physical addresses is necessary for two reasons: shared data or segments may have different virtual addresses but same physical addresses; lower level caches (such as L-2 or Last Level Caches) use physical addresses. We are exploring the use of more complex, full system simulators to translate virtual addresses generated by Gleipnir to physical addresses used by a running thread.

## Acknowledgment

This work is made possible in part by support from the NSF Net-Centric Industry/University Research Center, a grant from Advanced Micro Devices, and ORNL summer internship support. Computational resources were provided by UNT's High Performance Computing Initiative, a project of Academic Computing and User Services within the UNT computing and Information Technology Center. The name Gleipnir is taken from Norse mythology, and refers to a deceptively strong ribbon that was used to restrain a vicious wolf called Fenrir. Our aim was to use Gleipnir as aid to help programmers restrain the memory hogs in applications.

## References

1. Agarwal, A., Pudar, S.D.: Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In: Proceedings of the 20th annual international

symposium on computer architecture. ISCA '93, New York, NY, USA, ACM (1993) 179–190

2. Givargis, T.: Improved indexing for cache miss reduction in embedded systems. In: Proceedings of the 40th annual Design Automation Conference. DAC '03, New York, NY, USA, ACM (2003) 875–880
3. Patel, K., Macii, E., Benini, L., Poncino, M.: Reducing cache misses by application-specific re-configurable indexing. In: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design. ICCAD '04, Washington, DC, USA, IEEE Computer Society (2004) 125–130
4. Kharbutli, M., Irwin, K., Solihin, Y., Lee, J.: Using prime numbers for cache indexing to eliminate conflict misses. In: Proceedings of the 10th International Symposium on High Performance Computer Architecture. HPCA '04, Washington, DC, USA, IEEE Computer Society (2004) 288–
5. Peir, J.K., Lee, Y., Hsu, W.W.: Capturing dynamic memory reference behavior with adaptive cache topology. SIGPLAN Not. **33**(11) (October 1998) 240–250
6. Jula, A., Rauchwerger, L.: How to focus on memory allocation strategies. Tech Rept TR 07-003 (May 1999)
7. Chilimbi, T.: Making pointer-based data structures cache conscious (2000)
8. Kulkarni, C., Ghez, C., Miranda, M., Catthoor, F., De Man, H.: Cache conscious data layout organization fo r conflict miss reduction in embedded multimedia applications. Computers, IEEE Transactions on **54**(1) (jan 2005) 76 – 81
9. Calder, B., Krintz, C., John, S., Austin, T.: Cache-conscious data placement. In: in Proceedings of the Eighth International Conference on Architectural Suppo rt for Programming Languages and Operating Systems. (1998) 139–149
10. Kavi, K., Nwachukwu, I., Fawibe, A.: A comparative analysis of performance improvement schemes for cache memories. Computers & Electrical Engineering **38**(2) (2012) 243–257
11. Naz, A., Kavi, K., Rezaei, M., Li, W.: Making a case for split data caches for embedded applications. SIGARCH Comput. Archit. News **34**(1) (September 2005) 19–26
12. Zhang, C.: Balanced cache: Reducing conflict misses of direct-mapped caches. In: In Proc. of the 33rd Annual International Symposium on Computer Architecture, IEEE Computer Society (2006) 155–166
13. Janjusic, T., Kavi, K.M., Potter, B.: International conference on computational science, iccs 2011 gleipnir: A memory analysis tool. Procedia CS **4** (2011) 2058–2067
14. Jan Edler, M.D.H.: Dineroiv trace-driven uniprocessor cache simulator
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42** (June 2007) 89–100
16. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: Mibench: A free, commercially representative embedded benchmark suite. In: Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. (dec. 2001) 3 – 14