OPTIMIZING CACHE EFFICIENCY BY SIMULATION DRIVEN AUTOMATIC PADDING

Marco Höbbel, Thomas Rauber, Carsten Scholtes
Fachgruppe Informatik
Universität Bayreuth
Universitätsstr. 30
95447 Bayreuth
{hoebbel,rauber,carsten.scholtes}@uni-bayreuth.de

ABSTRACT

We present a toolset to automatically optimize the cache efficiency of an arbitrary application by dynamically padding memory allocations. The toolset is also suitable to guide manual optimizations. Histograms are used to evaluate cache simulations of memory traces of the applications considered. A general algorithm is presented that calculates optimized pad sets based on the information contained in the histograms. These pad sets can then be used to optimize further runs of the applications examined. Experiments show that the cache hit rates of the modified applications are considerably increased.

Keywords: *cache optimization, dynamic padding, memory tra ces, performance visualization*

I. INTRODUCTION

For most programs, the execution time should be as short as possible. Especially for computation intensive applications a good runtime efficiency minimizes the throughput time, thereby preserving computation resources for other pending or scheduled applications. Scientific problem solvers, for example, are mostly time and data intensive in nature. Many mathematical computations like vector-matrix or matrix-matrix multiplications and additions in multiphase, iterative or cyclic algorithms refer to user data in problem specific access patterns. For regular applications, it can be expected that repeated program executions exhibit a similar memory access pattern. Many data intensive applications therefore benefit from a high memory bandwidth which in modern architectures is supported by supplying a multi-level cache hierarchy. The efficiency of caching depends strongly on temporal or spatial reuse, so potential conflicts should be avoided. Different techniques are applied in order to ensure that the fastest cache level in the memory hierarchy is addressed first. Padding is one such approach and has been successfully applied for many high performance applications. Unfortunately, most of these optimization techniques are problem specific and difficult to adapt for general use. Additionally to a high experience of the programmer, a deep analysis of the program and its access pattern is required to become aware of the relevant effects and to finally optimize cache usage. Due to the immense effort associated with this approach, often only post-programming optimizations by the compiler or the runtime system are employed for gaining high efficiency with minimal effort.

In order to support analyzing code sections optimized by hand and runtime optimizations of applications without having their source code, a trace based acquisition of data for analyzing the application's memory access pattern is suitable. We present such an approach in this paper. The advantage of the approach is (1) that it can be applied also for programs for which the source code is unknown and (2) that the programmer does not have to apply low-level code optimizations to obtain a good overall performance. The approach is based on histograms which are data structures having the potential to hold the necessary information like access patterns acquired during a first phase tracing program run. Based on the histograms of an application, a second phase computes a memory aligning pad set of all or at least the major cache impacting references which leads to a better overall performance. The generated optimal offsets can then be applied by a compiler to create an optimized binary code. In the case of only having a binary executable, an enhanced runtime system can intercept memory allocation operations and patch them to provide the optimized padding for the following program runs.

The paper's objective is the presentation of a post-compilation, performance optimizing and visualizing toolset, which is suitable for general use, even in the absence of the source code of the application to be optimized. In section II, we introduce the notation used in the rest of the paper. In section III we describe the optimization method. It is subdivided into three parts concerning the trace driven data acquisition (III-A), the histogram types describing conflicts (III-B) and the automated optimization algorithm (III-C). In section IV we present measured results for different example applications before we discuss related work and conclude our paper.

II. TERMINOLOGY

The memory interface of the execution platform is modeled as follows: The cache [Handy, 1998] is subdivided into *cache lines*, which represent the minimal amount of data transferable between cache and main memory. The main memory is subdivided into *memory lines* of the same size S_{CL}, which are mapped round-robin onto the cache lines. An access which can be satisfied directly from the cache is called a (cache) *hit* and takes a latency

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

of t_{L_H} CPU cycles. If the cache does not contain a copy of the memory line to be accessed, the access is called a (cache) *miss* and results in a maximal stalling time of t_{L_M} cycles for validating by accessing the main memory. An access to a memory line that has been accessed before is called a *reuse*. The last access to the reused memory line before the reuse is called the *source access* of the reuse. The time between source access and reuse is called the *reuse distance* of the reuse. The set of accesses occurring within the reuse distance of a reuse, thus, potentially replacing the memory line to be reused, is called *interference* of the reuse.
Many cache hardware platforms do not only validate the cache line just accessed, but also preload the next k ones in order to prevent latency cycles by adopting an anticipating early validation strategy [Oren, 2000]. A simple *k-line* prefetching strategy with $k = 1$ tries to also load the line in sequence of the last two accesses into the cache
A *reference* is a contiguous chunk of memory containing elements accessed by the application. From the address bus' point of view, each reference R is classified by its base address $base\_addr_R$ (the first address of the chunk) and the size of its elements $size_R$. For analyzing, we collect all addresses of the accesses of an array reference and store an extra attribute that counts the number $length_R$ of array elements contained. The program/data *trace* T is the set of all accessed data- and instruction-bus addresses. It contains all collected accesses in the form of tuples $ma = (time, addr, acctype) \in T$ specifying $time \in [0,$ runtime in cycles$]$ in absolute system cycles, the accessed address $addr \in [0,$ virtual memory size$)$ of the reference and the access type $acctype$, denoted by either *r* (data read), *w* (data write), *d* (data read or write) or *i* (instruction read).
Furthermore, a discriminator $\mathbf{Ref}^{acctype}_{refname}$ can be applied to T to extract the specific accesses of type $acctype$ of the reference $refname$:

$$\begin{aligned} Ref^t_R := \{ & ma = (time, adr, type) : \\ & adr \geq baseadr_R, ma \in T, type = t \\ & adr < baseadr_R + size_R \cdot length_R \} \subset T \end{aligned}$$

For simplicity, Ref^*_R includes all traced tuples of reference R. If, for example, only the data bus is used, it is $Ref^*_R = Ref^d_R$.

III. METHOD

A. Data acquisition

In a first phase, a trace of a fixed problem size is generated by capturing the addresses and properties of all memory accesses. Different tools can be used to generate such traces:

QPT: On SPARC [Mauro and McDougall, 2001], [SPARC, 2000] architectures, for example, there exists the QPT [Larus, 1993] toolset. It distinguishes between data read, data write and instruction read accesses. These traces are useful for binary level post-compilation optimizations. Such optimizations will rely on all the runtime information caught. They will be realized by manipulating the original binary to use dedicated dynamic memory allocations.

SPD: Given the source code of the application to be optimized, there is another method for acquiring trace data that supports a deeper analysis. It is based on **s**elf **p**rotocoled **d**atatypes (SPD) that we developed and that can be used instead of the original data types. Tracing with these is helpful for examining and refining parts of the whole program by focusing on only the basic conflicting scenarios within the source code. Additional access attributes like the name of the triggering reference (*refname*) and its data type (*field-type*) can be extracted without changing the binary used to produce the trace. These attributes may be taken into account by future optimization strategies like, for example, a symbolic reference analysis.

Other: Due to the availability of the GNU tool *gdb* on many architectures, it is interesting to use it for data acquisition by simulated program runs. Binaries (with symbol table) include valuable symbolic information helpful for generating more transparent analyses.
Another pure simulation tool *Simics* [Magnusson et al., 2002], [Magnusson et al., 1998] is useful for address trace generation with the ability to focus on special application bounded addresses only.

For these and other tools it is mandatory that the trace generation is not corrupting inter-reference correlations. Apart from analyzing given applications, the tracing tools are also useful to verify the benefits of optimizations.

B. Histograms

During the trace run of the program to be analyzed, the generated access pattern has to be stored in a way that conserves information on which the later optimization phase depends. Our simulation based histogram illustrations expose patterns in an intuitively understandable manner. A directed extraction of the data stored in the first phase supports the final automatic optimization step. Furthermore, the intuitive understandability of graphically visualized histograms inspires the invention of new types of histograms capturing additional information suitable to support a more target oriented or better performing analysis.
For simplicity, we consider in the following only data accesses T^d_* (both, reading and writing) and no instruction reading accesses. Our histograms count accesses to address offsets or address differences of accesses. They hold counts for up to H_{size} consecutive such address values. H_{size} will match the size S_C of the cache of the target platform. Different types of histograms are employed:

SRH – single reference histogram: The stored data of the first histogram type ${}_SH_R$ is concerned with a single reference R only. For an address $baseadr_R + i$ the value ${}_SH_R(i)$ counts the number of memory ac-

cesses to the address offset i of reference R, such that $\sum_{i=0}^{H_{size}-1} {}_SH_R(i) = |Ref_R^d|$. The latter represents the *significance* of the reference R in comparison to the other references in the program trace. The generated pattern stored in the histogram is evaluated by accumulating the offsets from the base address $baseadr_R$. The histogram ${}_SH_R$ is then defined as the set of all pairs of offsets i and their corresponding access count ${}_SH_R(i)$:

$$\begin{aligned} {}_SH_R(i) &:= |\{ma = (time, adr, type) : \\ &\quad ma \in Ref_R^d, \\ &\quad i = (adr - baseadr_R)\%H_{size}\}| \end{aligned}$$

$$\begin{aligned} {}_SH_R &:= \{(i, {}_S H_R(i)) : \\ &\quad i \in [0, H_{size})\} \end{aligned}$$

For scalar 0-dimensional types of references, the histograms degenerate to simple counters.

SDH – self distance histogram: A self distance histogram ${}_DH_R$ refers to a single reference R, too, but accumulates the distances of consecutive accesses to R. We first introduce the predecessor $pred_{R_2}(ma) \in Ref_{R_2}^d$ of an access $ma = (t_a, a_a, t) \in Ref_{R_1}^t$ in order to then define ${}_DH_R(i)$ for an address distance i:

$$\begin{aligned} pred_{R_2}(ma) &:= (t_b, a_b, t) \in Ref_{R_2}^t \text{ with} \\ t_b &= \max\{time : \\ &\quad \exists\, (time, a, t) \in Ref_{R_2}^t, \\ &\quad time < t_a\} \end{aligned}$$

$$\begin{aligned} {}_DH_R(i) &:= |\{ma = (t_a, a_a, d) \in Ref_R^d : \\ &\quad \exists\, (t_b, a_b, d) = pred_R(ma), \\ &\quad (a_a - a_b)\%H_{size} = i\}| \end{aligned}$$

The information extracted by this kind of histogram describes the step increment of sequential memory accesses. Other references with similar step increment patterns can be padded in order to avoid thrashing constellations. Furthermore, given a sequential access pattern, which can be observed for many references, the histogram allows us to detect the prefetch distance and to estimate the resulting usage efficiency for each cache line fetched. Larger distances imply a sparser usage. This information can be used to resolve padding conflicts with other references by favoring references with more efficient cache line usage.

PDH – pairwise distance histogram: This type of histogram calculates distances like SDH, but this time, the accesses refer to the consecutive accesses between two distinguished references R_1 and R_2:

$$\begin{aligned} {}_PH_{R_1,R_2}(i) := |\{&ma = (t_a, a_a, d) \in Ref_{R_1}^d : \\ &\exists\, (t_b, a_b, d) = pred_{R_2}(ma), \\ &((a_a - baseadr_{R_1}) - \\ &(a_b - baseadr_{R_2}))\%H_{size} = i\}| \end{aligned}$$

The histogram ${}_PH_{R_1,R_2}$ highlights distances conflicting spatially. To avoid the problem of mutual thrashing, all significant distances in ${}_PH_{R_1,R_2}$ should be excluded in the later pad set.

```
sList =sort( _S H*^d by | _S H^d|)
_g H = H^d_stack
FOR H ∈ sList DO
      offsets[H] = f_min(_g H, H, offsets)
      _g H = _g H ⊕ (H ≫ offsets[H])
DONE
```

Figure 1: Optimization Algorithm

Histograms support a number of basic operations useful during the optimization. Two histograms H_1 and H_2 can be *merged* to a new histogram $H = H_1 \oplus H_2$ which is defined a follows:

$$\forall_{i=0}^{H_{size}-1} H(i) = H_1(i) + H_2(i)$$

A histogram H can be *rolled* by an offset o to yield a new histogram $H' = H \gg o$ which is defined as follows:

$$\forall_{i=0}^{H_{size}-1} H'((i+o)\%H_{size}) = H(i)$$

The *significance* $|H|$ of a histogram H is defined as the number of the accesses contained:

$$|H| := \sum_{i=0}^{H_{size}-1} H(i)$$

In the following, these operations and constructs are used to formulate the optimization algorithm.

C. Optimization

The third and final phase is the optimization. It uses the different types of histogram data generated to determine a pad set for a later cache optimized program execution. The algorithm is based on a greedy strategy recognizing the patterns the histograms describe. Appropriate heuristics consider the conflict potential and find a padding with minimal miss potential.

The algorithm we propose, as outlined in Figure 1, is straightforward and tries to simplify the complex problem of finding the best fitting overall solution by using reference and inter-reference specific histogram overlaying. To do so, (1) it applies the greedy paradigm to a sorted list of histograms which are padded with a "highest significance first" strategy. Then, the reference chosen to be padded is shifted by an offset that offers the best overlay with a global histogram ${}_gH$. This offset is the current reference's relative pad. It is determined (2) by a minimization function $f_{min}({}_gH, H, \text{offsets})$. The global histogram ${}_gH$ represents the conflict potential of the references padded so far. It is initialized with the histogram pertaining to special references like the stack area that cannot be padded easily. Each time a new offset has been determined, the global histogram is updated by merging it with the current histogram rolled by the amount of its offset. Frequently, there exists a set O_{min} of multiple offsets with a minimal or near minimal conflict potential according to f_{min}. These alternatives offer flexibility when optimizing simultaneously for further targets like,

for example, for efficient use of more than one level of a cache hierarchy.

Equipartition: A simple, intuitive approach is trying to make use of the whole cache. The function f_{min} in charge tries to achieve a uniform distribution of cache accesses by padding all references according to their accesses described in the *SRH* histograms. In each step of the algorithm, the current histogram H of a single reference and the current global histogram ${}_gH$ are merged with an offset found by a normal deviation guided function f_{min}. We first define the histogram ${}_jH$ as the global histogram after merging it with the current histogram H at offset j. Then, we define s_j as a measure for the deviation to be expected for offset j. The set S is defined as the set of values s_j for all available offsets j. The set O_{min} contains the offsets corresponding to a minimal or near minimal deviation. The parameter $\epsilon \geq 0$ controls the accepted range of results from which f_{min} chooses its result.

$$
\begin{aligned}
{}_jH &:= {}_gH \oplus (H \gg j) \\
s_j &:= \tfrac{1}{H_{size}} \textstyle\sum_{i=0}^{H_{size}-1} ({}_jH(i) - \tfrac{1}{H_{size}} |{}_jH|)^2 \\
S &:= \{s_j : j \in [0, H_{size})\} \\
O_{min} &:= \{i : s_i \leq \epsilon + min\ S\}
\end{aligned}
$$

This strategy is especially useful for applications whose memory footprint is smaller or not much larger than the cache size.

Inter-reference distance: Each non zero value in a PDH difference histogram points out a potential conflict for the pair of references concerned. The histogram bars weight the conflict distances between the two references. A single bar's height indicates the conflict potential for the bar's offset. Thus, in order to avoid mutual cache thrashing the difference in the padding offsets for the two references concerned, is chosen to correspond to a bar as low as possible considering that this optimization has to be done simultaneously for all PDH-histograms of the current reference with the references already padded in the global scope.

Self distance restrictions: In the cases of multiple similarly optimal offsets determined by other strategies, the histogram type *SDH* can additionally optimize for k-line prefetching. Prefetching is very susceptible to memory bandwidth, which can be lowered by padding references with similar self distances, as described by the SDHs, to cache regions avoiding continued thrashing.

After determining the offsets forming the runtime pad set, the memory allocation can be further optimized without influencing the padding offsets to minimize the total memory usage. This optimization is not in the scope of this paper. It resembles the enlarged backpack problem for a multiple count of packs.
Finally the optimization results are stored in a single file associated with the executables name.

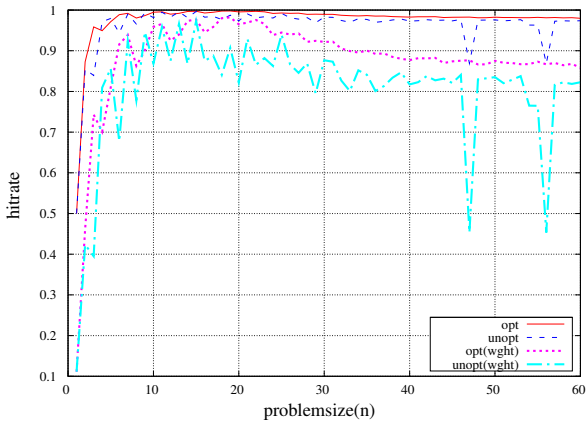


Figure 2: Cache Hit Rate for LU Decomposition on a 4k Direct Mapped Cache with Line Size 32

D. Optimized Execution

The final target of the considerations above is the optimized cache aware execution of the traced program. We achieve it with one of two options:

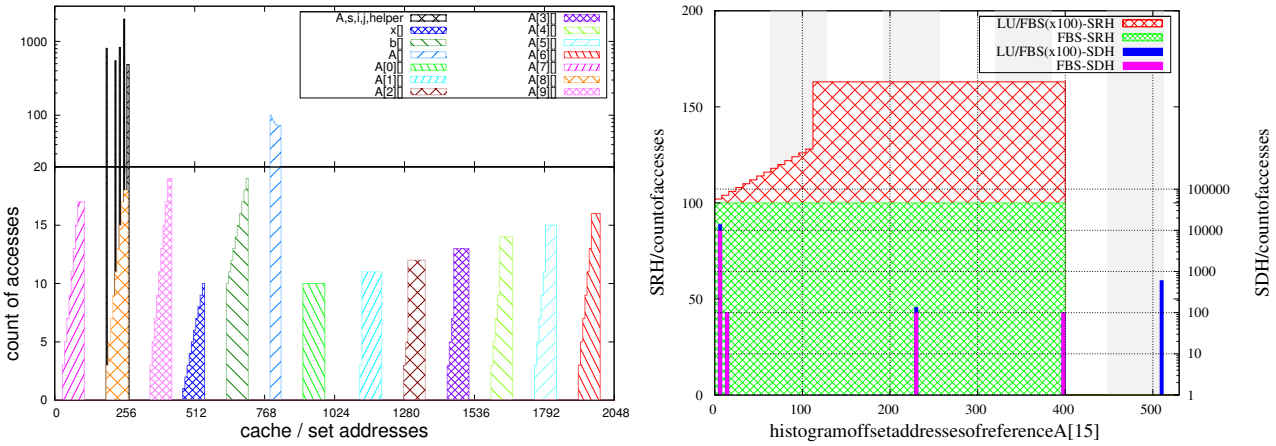
ld_preload: In order to intercept the original memory allocation operations associated to individual references, we have to catch and to patch all `malloc`, `free` and `realloc` system calls during the runtime of the program to be optimized with the help of the pre-determined pad set. A simple but effective way of doing this, is to manipulate the internal hooks of the memory management system. Therefore, a preloaded code overriding the original allocations was written, which sets up the pad offset of each dynamically allocated reference. The original program code does not need to be aware of these circumstances, in fact, the base address is shifted to the optimized offset but from the application's perspective it appears as a normal memory allocation.

Compiler, Datatypes: Having the source of the application to be optimized, the compiler itself is capable to shift the references to the pre-determined offsets, as well. In this case, all pads are already set and fixed up in the compilation. Besides, this source level patching offers zero runtime overhead for accessing the pre-determined padded structures by simple base address shifts which have to be done in the unoptimized case, too.

In our testing environment we actually do the padding by applying the pad set to the self protocoling data types.

IV. EXPERIMENTAL RESULTS

To show the usability of the concept of automated simulation driven padding, we chose the often used matrix data structure and, as a typically applied operation, the matrix multiplication. The multidimensional structures are made up of separate arrays in order to keep the opportunity for reordering and to maximize the number of references that could be padded. This assumption reflects the reality of often used row or column major data types in high level programming languages. C-programs should



(a) (b)

Figure 3: (a) Histogram for LU Decomposition with n=10 (b) SRH, SDH of the Row Reference A[15]

use row-major memory alignment for efficient memory access, but not all programs are optimized in this way. For this reason, non language conform data to memory mappings are interesting, too, for showing that padding can support a better cache performance for such programs, as well.

In the following, we present the results of examining and optimizing a few different applications.

A. LU decomposition to solve linear equation systems

We examine data intensive solvers as the *Gauß*-algorithm in order to design target oriented optimization heuristics. In Figure 3(a) the first histogram type *SRH* shows the access patterns of each reference with their original pads. There are graphs for the rows A_i of matrix A, its row pointers $A[]$, the vectors x, b and the stack segment including the local variables as well as loop iterators, accumulating variables and the pointer to the matrix A. The chosen problem size of $n = 10$ allows to conveniently display at once all the access patterns resulting from solving a linear system of equations.

In order to solve the problem $Ax^{(i)} = b^{(i)}$ several times with a modified vector $b^{(i)}$ and a static coefficient matrix $A \in R^{n,n}$ we can decompose A into a lower triangular matrix L and an upper triangular matrix U with $LU = A$. Both of the triangular matrices are stored in the original places of the former quadratic coefficient matrix A. U is formed by applying the Gauß elimination algorithm to A while L holds the corresponding elimination factors. The computation of L and U takes place only once and has an asymptotic complexity of $O(n^3)$. The forward substitution $Ly = b^{(i)}$ derives y from the current vector $b^{(i)}$ and the backward substitution $Ux^{(i)} = y$ solves the proper problem. Both operations are of complexity $O(n^2)$ and address the same references the LU decomposition already accessed.

Figure 3(b) on the right side shows the histograms *SRH* and *SDH* of the single row A_{15}'s accesses in one diagram for solving the problem $Ax^{(i)} = b^{(i)}$ of the size $n = 50$ for 100 different vectors $b^{(i)}$. The pattern for ${}_SH_{A_{15}}$ is as expected and mixes phases one and two by counting all accesses. The overlayed (right axis) histogram ${}_DH_{A_{15}}$, depicts the difference patterns split according to the two sequential phases of decomposition and solution. The pattern ${}_DH$ of a matrix row generates, in addition to the linear inner loop sequential accesses, spikes for reverse directed accesses for every line rewind enforced by the outer loop of the decomposition. The 100 forward and backward substitutions are computed in place by altering the elements of vector b. In the difference pattern histogram, they show a behavior similar to that of the LU decomposition.

Figure 2 compares the performances before and after optimization for different system sizes n. One measurement comprises one LU decomposition and 100 repeated solutions. The performance of the optimized versions is consistently higher and the results behave much more stable than with the original, straightforward padding provided by the memory allocation system.

B. Matrix-Matrix Multiplication

The results for another typical problem of an optimized single matrix-matrix multiplication $C = A \cdot B$ for different system sizes n shown in Figure 4 are padded into a 4k direct mapped cache with line size 32 and hit / miss latencies of $t_{L_H} = 1$ and $t_{L_M} = 7$, respectively. The matrices $A, B, C \in R^{n \times n}$ can be stored in row- or column-major order. For the experiment's results presented in Figure 4(a) a unique row-major ordering for all three matrices was chosen (row-row-row ordering). For small n, compulsory misses are dominant and memory bandwidth is poorly used with only partially accessed cache lines. For $n < 14$ the whole problem can be cache contained although the unoptimized allocation scheme arranges references' footprints usually cyclically within the cache issuing larger chunks of memory aligned to memory paragraphs, compare Figure 3(a). Our current padding heuristic compacts all references for minimal, but effective

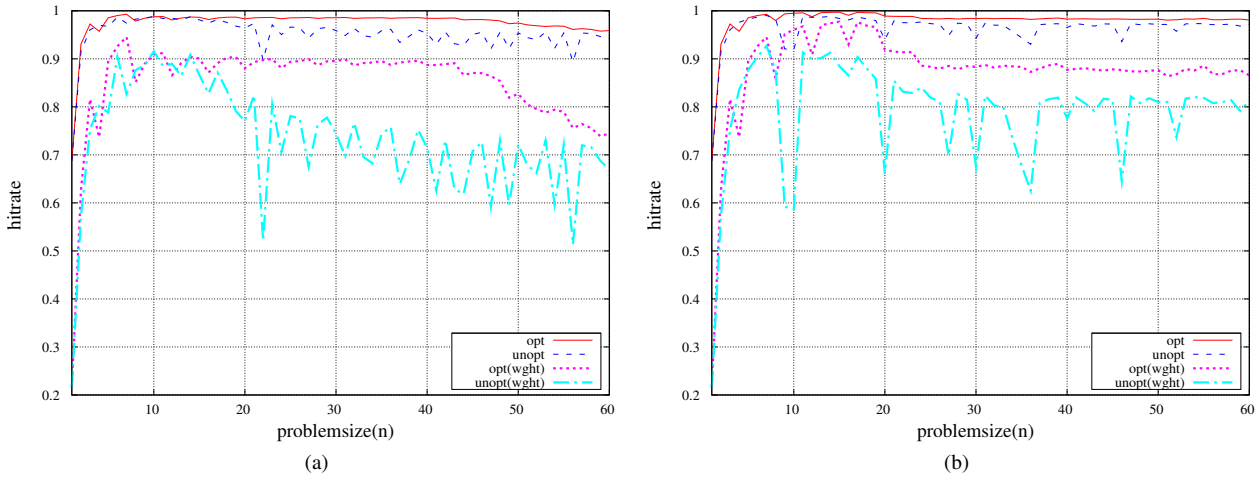


Figure 4: Matrix-Matrix Multiplication on a 4k Directed Mapped Cache with Line Size 32; (a) Exclusive Row, (b) Row-Column Ordered

cache usage. With increasing problem size, the original program runs suffer early of slight but continuous degeneration of cache performance because the probability of thrashing grows with n. Badly chosen memory alignments drastically impact the over-all hit rate quite often. The *weighted* plots in the figures discussed give an impression of the runtime effect based on the efficiency of the program's data cache usage. Given the numbers h of hits, m of misses and the latencies t_{L_M} and t_{L_H}, we estimate the total sum of cycles used to access memory as $h \cdot t_{L_H} + m \cdot t_{L_M}$. The weighted hit rate hr_w is then defined as follows:

$$hr_w = \frac{h \cdot t_{L_H}}{h \cdot t_{L_H} + m \cdot t_{L_M}}$$

Figure 4(b) depicts the results measured for a more efficient implementation with row-column-row major memory alignment of the three matrices A, B and C, respectively. Due to the smaller number of potentially conflicting cache lines, the over-all performance is further increased compared to the row-row-row major memory ordering in Figure 3(a).

C. Runge-Kutta solver for ODEs

As an example dealing with different data structures and access patterns, we examined an application to solve ordinary differential equations (ODEs) with a Runge-Kutta algorithm. For certain problem sizes this application displayed unexpectedly long run times. Using our toolset we were able to identify a thrashing constellation in these configurations. An analysis of the access patterns enabled us to devise a new solver with a consistently improved memory behavior [Korch and Rauber, 2006].

V. RELATED WORK

Techniques for increasing the locality of memory references have been studied extensively. An important, purely analytical model for effects of loop transformations [Abella et al., 2002] partly combined with padding and tiling can evaluate cache performance using CMEs (Cache Miss Equations) [Ghosh et al., 1997], [Vera et al., 2004] for special and zoned loop constructs. Another approach [Scholtes, 2003], based on conflict classes, does so for the Cholesky factorization. For applications with seemingly irregular or complex access patterns, an alternative memory mapping can be applied by evaluating indices with the help of an easy to compute map function. The memory access patterns generated by these functions are designed to have a better cache performance but they are strictly problem specific [Coleman and McKinley, 1995]. The Morton-Ordering is such a more general approach optimizing the memory layout for matrix operations as proposed in [Thiyagalingam and Kelly, 2006]. Many scientific libraries like LAPACK [Anderson et al., 1999] are based on the BLAS (Basic Linear Algebra Subprograms) that can be considered as a de facto standard for linear vector matrix based numerical algorithms. ATLAS (Automatically Tuned Linear Algebra Software) [Whaley et al., 2001] provides an efficient implementation of BLAS routines, as well as a genetic [Vera et al., 2003] CME guided algorithm for detecting well performing tile sizes [Jin et al., 2001], [Rivera and Tseng, 1999] and pad offsets [Vera et al., 2002].

VI. CONCLUSION

To our knowledge we present a new approach to design a post compilation performance optimization tool set for general use without necessarily having the source code of the application to be optimized available. Additional tools are provided for analyzing graphically the data acquired in the first phase trace run and the results of the optimizations. One of these is an extensible cache simulation engine with a graphical front end supporting the issues of our optimization strategies. It is also suited for guiding a manual program optimization.

The proposed automated optimization algorithm uses the data previously acquired and stored into histograms describing access patterns. It balances the potential

of single references to cause cache conflicts. Measured results show a significantly improved cache performance. Unfortunate memory allocations producing excessive thrashing are avoided altogether.

For future, more sophisticated optimization heuristics accounting for multi-level caches, the different memory hierarchy latencies will be used to refine the prediction of miss penalties and, along with this, the automatic optimization. Furthermore we expect it to be possible to extrapolate the memory behavior of an application from a few small problem sizes to larger problem sizes by spreading the measured histograms accordingly. Our distance histograms currently disregard the time passed between two accesses whose distance is recorded. We plan to introduce a temporal component reflecting this time in order to provide a better base for optimizations.

REFERENCES

[Abella et al., 2002] Abella, J., Gonzàles, A., Llosa, J., and Vera, X. (2002). Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proceedings of the 2002 ICPP Workshops*, pages 568–577.

[Anderson et al., 1999] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarlin, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide, Third Edition*. Society for Industrial and Applied Mathematics.

[Coleman and McKinley, 1995] Coleman, S. and McKinley, K. S. (1995). Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95), SIGPLAN Notices*, La Jolla, CA.

[Ghosh et al., 1997] Ghosh, S., Martonosi, M., and Malik, S. (1997). Cache miss equations: An analytical representation of cache misses. Workshop on Interaction between Compilers and Computer Architectures, Third International Symposium on High-Performance Architectures (HPCA-3).

[Handy, 1998] Handy, J. (1998). *the Cache Memory book*. Academic Press, Inc., 2nd edition. THE authoritative reference on cache design.

[Jin et al., 2001] Jin, G., Mellor-Crummey, J., and Fowler, R. (2001). Increasing temporal locality with skewing and recursive blocking. In *SC2001: High Performance Networking and Computing*. ACM Press and IEEE Computer Society Press. CD-ROM.

[Korch and Rauber, 2006] Korch, M. and Rauber, T. (2006). Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. *J. Par. Distr. Comp.*, 6(3):444–468.

[Larus, 1993] Larus, J. R. (1993). Efficient program tracing. In *Proceedings of IEEE Computer*, pages 52–61.

[Magnusson et al., 2002] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Hgberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: A full system simulation platform. *Computer*, 35(2):50–58.

[Magnusson et al., 1998] Magnusson, P. S., Dahlgren, F., Grahn, H., Karlsson, M., Larsson, F., Lundholm, F., Moestedt, A., Nilsson, J., Stenstrøm, P., and Werner, B. (1998). SimICS/sun4m: A virtual workstation. In *Usenix Annual Technical Conference*, New Orleans, Lousiana.

[Mauro and McDougall, 2001] Mauro, J. and McDougall, R. (2001). *SOLARIS Internals*. Sun Microsystems Press - A Prentice Hall Title.

[Oren, 2000] Oren, N. (2000). A survey of prefetching techniques. Technical Report number CS-2000-10, University of the Witwatersrand, Johannesburg, South Africa.

[Rivera and Tseng, 1999] Rivera, G. and Tseng, C.-W. (1999). A comparison of compiler tiling algorithms. In Proceedings of the 8th International Conference on Compiler Constuction (CC'99).

[Scholtes, 2003] Scholtes, C. (2003). *Abschätzung der Fehlzugriffe bei dünn besetzten Matrixoperationen auf Architekturen mit einem direkt mapped Cache*. Dissertation.

[SPARC, 2000] SPARC (2000). *The SPARC Architecture Manual Version 9*. SPARC International, Inc.

[Thiyagalingam and Kelly, 2006] Thiyagalingam, J. and Kelly, P. H. J. (2006). Is morton layout competitive for large two-dimensional arrays ? *Concurr. Comput. : Pract. Exper.*, 18(11):1509–1539.

[Vera et al., 2003] Vera, X., Abella, J., Gonzalez, A., and Llosa, J. (2003). Optimizing program locality through cmes and gas. *pact*, 00:68.

[Vera et al., 2004] Vera, X., Bermudo, N., Llosa, J., and Gonzalez, A. (2004). A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(2):263–300.

[Vera et al., 2002] Vera, X., Llosa, J., and González, A. (2002). Near-optimal padding for removing conflict misses. In Pugh, W. and Tseng, C.-W., editors, *LCPC*, volume 2481 of *Lecture Notes in Computer Science*, pages 329–343. Springer.

[Whaley et al., 2001] Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.