A Data Cache with Dynamic Mapping

Paolo D'Alberto, Alexandru Nicolau, and Alexander Veidenbaum

Department of Computer Science
University of California, Irvine[⋆]
{paolo,nicolau,alexv}@ics.uci.edu

Abstract. Dynamic Mapping is an approach to cope with a loss of performance due to cache interference and to improve performance predictability of blocked algorithms for modern architectures. An example is matrix multiply: tiling matrix multiply for a data cache of 16KB using optimal tiles size achieves an average data-cache miss rate of 3%, but with peaks of 16% due to interference.

Dynamic Mapping is a software-hardware approach for which the mapping in cache is determined at compile time, by manipulating the address used by the data cache. The reduction in the cache misses translates into a 2-fold speed-up for matrix multiply and FFT by eliminating data-cache miss spikes.

Dynamic mapping has the same goal as other proposed approaches, but it determines the cache mapping before issuing a load. It uses the computational power of the processor - instead of the memory controller or the data cache mapping - and it has no effect on the access time in memory and cache. It is an approach combining several concepts, such as non-standard cache mapping functions and data layout reorganization and, potentially, without any overhead.

1 Introduction

The increasing gap between memory access latency time and CPU clock cycle makes it extremely hard to feed a processor with useful instructions or data. This problem is exacerbated in multiprocessor systems and distributed systems because of extremely high demand of data and instructions, and because of communication through relatively slow devices. To avoid CPU stalls due to data and instruction starvation, several approaches have been proposed. We can divide them in three broad and, somewhat arbitrary, classes.[1]

Memory hierarchies and utilization bounds: we consider in this class hardware implementations of memory systems, interconnection of memory systems and performance bounds for families of algorithms on architectures with

[⋆] This work is supported in part by NSF, Automated Application Development Environment for Future Complex Heterogeneous Computing Platforms. Contacts:

[1] Arbitrary because approaches in different classes may share several goals and properties, and incomplete because the literature is so rich that is difficult to keep record.

memory hierarchy. For example of hardware implementation, Smith offers a survey [26] on caches, an excellent start reference for uniprocessor systems is Hennesy and Patterson [16]. For examples of performance bounds, Hong and Kung propose lower bounds for simple memory architectures with 1-level of cache [17], Aggarwal et al. and others researchers generalize the bounds to multiple level of caches [3, 4, 29, 30, 2].

Application reorganization: in this class we consider projects related to the trade-off between portability and performance across different architectures with multiple level of caches. For example, we find in this class optimizations by code reorganization through tiling [1, 32], by selective copying [19], by tailoring the application at installation time [31, 22], or by utilization of memory hierarchy independent optimizations [7].

Hardware-software adaptive approaches: we consider in this class on-the-fly hardware and software adaptations. An example of at-run-time hardware adaptation is cache-associativity adaptation, and an example of the software adaptation is the at-run-time reorganization of registers allocation to reduce register file power dissipation, [10, 15, 6]. Another example of algorithm adaptation is the work by Gatlin et al. [13], where data-copy strategies are applied to exploit cache locality.

Our approach is a software-hardware approach and we propose it to solve the problem of cache interference in blocked algorithms. Blocked algorithms, such as blocked Matrix multiply and FFT, achieve good cache performance on *average*; however, we notice a quite erratic cache behavior on *individual* input sets - due to cache interference. We propose an approach to minimize cache misses due to cache interference changing the cache mapping for some memory references dynamically.

We aim at the optimization of blocked algorithms because of their data locality properties. A blocked algorithm applies a divide and conquer approach: a blocked algorithm divides the problem in smaller problems and it solves them locally. The advantage is clear when the decomposition of the problem exploits the available parallelism and the memory hierarchy (i.e., the data of every sub-problem fit the local memory or the data cache of a processor). The divide and conquer approach improves data reuse and it reduces communication among processors as well as between cache and memory. Avoiding data interference in cache is the last, and final, step to exploit data locality fully.

We enforce the problem by an example quantitatively. We implement an optimal blocked implementation of matrix multiply for an architecture with a directed mapped data cache of size 16KB. We opt for matrices stored in row-major format, which are used commonly. We design the algorithm with no pre-fetching - pre-fetching hides the latency but does not reduce cache misses. The blocked algorithm can be the result of tiling exploiting cache locality on a uniprocessor system, or the result of a parallelizing compiler for shared-memory multiprocessor systems (or both). We achieve on a uniprocessor system an average 3% data cache miss rate. The average cache miss is close to the optimal cache performance (roughly 0.5%, [17]). When we observe only the cache performance for square matrices of size $2^k \times 2^k$, the miss rate is higher than the average. In particular for square matrix of size 2048×2048, the data cache miss rate is 16% and this is due to data cache interference only.

We propose a software-hardware approach to remove data cache miss spikes, changing the cache mapping only when needed, our approach is called **Dynamic Mapping**:

1. We produce a blocked algorithm, either by tiling of a loop nest or by a recursive implementation, so that we maximize temporal locality for the cache level we are interested into.
2. The blocked algorithm has each elementary block computation (i.e., loop tile) accessing rectangular tiles of the data (i.e., tile of matrices).
3. For each memory reference in the elementary block computation, we determine a physical address and an alternative - and unique - address, **twin address**. The physical address is used to map the element in memory; the twin address is used to map the element in cache (the details, how to determine and use twin addresses are explained for an example in Section 2.1) .

   The twin address space does not need to be physically present and, in practice, the twin space is larger than the physical space. A 64bit-register can address $2^{64} \sim 1 * 10^{19}$ bytes of memory, relatively few bytes are physically available.
4. The physical address is used whenever there is a miss in cache to access the second level of cache or memory; we assume the cache is physical tagged, and we can modify the processor and the load queue for our purpose (we shall give more details in Section 2.3).

We can apply our approach to a family of blocked algorithms. A blocked algorithm in this family has the following properties:

– **Fixed decomposition.** The algorithm does not change at runtime.
– **Convexity.** The algorithm solve a problem by dividing the problem in smaller problems. The problem and each sub-problem work on data with similar shape and properties; that is, if we solve a problem on square matrices, every subproblem works on square tiles - with the property that elements in different tiles cannot use the same cache line.
– **Total reuse.** If a data is used by two different subproblems, both use the common data consistently: every element in a tile is always mapped with the same twin address to the same location in cache.
– **Leaf Size.** The decomposition stops as soon as each sub-problem fits a target cache. We can copy all tiles, without overlapping, in a common space no larger than the cache size.

Our approach analyzes the application at compile time. The analysis is used to determine a data cache mapping to minimize data cache misses. Such mapping is introduced in the code as affine functions. The affine functions are computed at run time and the results used as alternative addresses. These addresses are used to map data in cache. It has the same effect to have the data layout reorganized in memory at runtime [11], using the computational power of the processor, with no data movement [19], no overhead or extra accesses. It is in practice a data mapping in cache that uses only partial information about the operations schedule (for example of cache mapping using a DAG computation see Bilardi and Preparata [2] and D'Alberto [8])

Dynamic mapping differs from IMPULSE project [18, 24], which introduces a new memory controller leaving the memory hierarchy untouched. IMPULSE supports a *configurable physical address mapping* and *pre-fetching at memory controller*. Our approach is simpler in the sense that it does not require either an operating system layer and any changes to the memory controller. The cache mapping is defined completely by the application, and it can be driven automatically by a compiler.

Dynamic mapping does not need any profile-based approach or dynamic computation changes [13]; it improves portability and let the developer focus on the solution of the original problem. We differ from Johnson et al. work [20, 21], our approach does not keep any track of memory references using dedicated hardware, the reference pattern is recognized statically.

Dynamic mapping does not change the physical data cache mapping [25, 14, 33] and, potentially, it has no increase of data cache latency.

Dynamic mapping is not a bypass technique: we are able to exploit more efficiently data locality - for a level of cache - for algorithms that have data locality; the processor does not need bypass a cache entirely. Cache bypassing is an efficient technique designed to increase the bandwidth between processor and memory hierarchy. In general, cache bypassing increases traffic on larger caches, which are slower and more energy demanding (see processors as R5k), and it does not aim to reduce data cache misses. Furthermore, cache bypassing makes the design more complicated and suitable for a general-purpose and high-performance processor.

Dynamic mapping is a 1-1 mapping among spaces; therefore it assures cache mapping consistency for any loads and writes to/from the same memory location. Hardware verification approaches for stale-data in registers - used by processors-compilers that allow speculative loads; for example, IA64 microprocessor [9] - can be safely applied.

The paper is organized as follows. Our software-hardware approach is presented in Section 2. In Section 2.2 we apply our approach to recursive FFT algorithms. FFT does not satisfy the convexity property of the decomposition, but we can apply successfully dynamic mapping. In Section 2.3 we propose the potential architectural modifications to support our approach. We present the experimental setup and results in Section 3. Then we conclude in Section 4.

2 Dynamic Mapping

In this section we investigate a software-hardware approach to minimize data cache interferences for perfect loop nest with memory references expressed by affine function of the loop indexes. This is a common scenario, where other approaches have been presented and powerful analysis techniques can be applied; for example, the analysis techniques proposed by Ghosh et al. [27], by Clauss et al. [5], by D'Alberto et al. [28] or other approaches based on Omega Test [23], are worth to be applied even though they are time expensive.

In Section 2.1, we propose our approach in conjunction with tiling to perfect loop nests, and in Section 2.2, we show that the approach can be successfully applied to recursive algorithms as FFT, which does not satisfy the convexity property - because the algorithms cannot always access continuous elements.

2.1 Matrix Multiply

In matrix multiply, every memory reference in the loop body is determined by an affine function of loop indexes, for short, **index function** (note: scalar references, if any, are array references with constant index function). An index function determines an address used to access the memory and the cache at a certain loop iteration. The idea is to compute, in parallel to a regular index function, a **twin function**. The twin function is an affine function of the indexes and it maps a regular address to an alternative address space, or **shadow address**. The index function is used to access the memory; the twin function is used to access the cache. In practice, a compiler can determine the twin functions as result of an index function analysis and it can tailor the data cache mapping for each load in the inner loop.

Example 1. We consider square matrices of size $N \times N$.

```
for (i=0;i<N;i++)
  for (k=0;k<N;k++)
    for (j=0;j<N;j++)
       C[i][j] += A[i][k]*B[k][j];
```

The reference $A[i][k]$ is a constant in the inner loop and it has index function $A_0 + N * i + 1 * k + 0 * j$. The index function for $C[i][j]$ (respectively, $B[k][j]$) is $C_0 + N * i + 0 * k + 1 * j$ (respectively, $B_0 + 0 * i + N * k + 1 * j$).

Matrix multiply loop nest can be reorganized to exploit temporal locality.

Example 2. Let us tile the loop nest by square tiles of size $s \times s$; we assume that N is a multiple of s, and matrices are aligned to the line size, and s is a multiple of the line size L:

```
for (i=0;i<N/s; i++)
 for (j=0;j<N/s;j++)
  for (k=0;k<N/s;k++)
   for (kk=0;kk<s;kk++)
    for (ii=0;ii<s;ii++)
     for (jj=0;jj<s;jj++)
      C[i*s+ii][j*s+jj] += A[i*s+ii][k*s+kk]*B[k*s+kk][j*s+jj];
```

When $3s^2 < S$, we achieve $\frac{2}{sL}N^3 + \frac{N^2}{L}$ memory accesses (cache misses). Cache misses may be more, due to interference.

When we tile the loop as in Example 2, the index function can be described concisely by three vectors (or projections onto 1-dimensional space), one for each matrix: $\alpha_1 = [Ns, 0, s, 1, N, 0]$, $\beta_1 = [0, s, Ns, N, 0, 1]$ and $\gamma_1 = [Ns, N, 0, 0, N, 1]$. If we indicate with $\iota = [i, j, k, kk, ii, jj]^t$ an iteration in the loop nest, the index function for A, B and C are $\alpha_1 * \iota + A_0$, $\beta_1 * \iota + B_0$ and $\gamma_1 * \iota + C_0$.

When matrix $N = 2^n > S$, $S = 2^k$ and all matrices are stored continuously one after the other ($A_0 + N^2 * \ell = B_0$ and $B_0 + N^2 * \ell = C_0$), there is cross interference between references to different tiles (e.g., $\mathbf{C}[i * s + ii][j * s + jj]$ and $\mathbf{B}[k * s + kk][j * s + jj]$) and there is self interference between any two rows in the same tile (e.g., $A[i * s + \mathbf{ii}][k * s + kk]$ and $A[i * s + \mathbf{ii} + \mathbf{1}][k * s + kk]$).

When we tile the loop nest, we tile each matrix as well. Every tile is a square, as the matrix, and it has size $s \times s$. When matrices are aligned to the cache line (i.e.; $A_0\%L = B_0\%L = C_0\%L = 0$) and all tiles are aligned to the cache line (i.e., $s\%L = 0$ and $N\%L = 0$), we can change the data-cache mapping for all memory reference safely.

An element in the 6-dimensional space is associated to a **twin element** in a 6-dimensional space. The difference is that we enforce the projection of a twin tile to be a convex space on a 1-dimensional space. The twin tile is stored continuously in memory - but the tile, with which is associated, needs not. Any two twin tiles are spaced at interval of S elements; so different twin tiles will be mapped into the same cache portion.

The twin function uses the following vectors: $\alpha_s = \left[\frac{N}{s}S, 0, S, 1, s, 0\right]$, $\beta_s = \left[0, S, \frac{N}{s}S, s, 0, 1\right]$ and $\gamma_s = \left[\frac{N}{s}S, S, 0, 0, s, 1\right]$. The twin functions for A, B and C are $\alpha_s * \iota$, $\beta_s * \iota + s^2$ and $\gamma_s * \iota + 2 * s^2$.

We consider in details the construction of $\alpha_s = \left[\frac{N}{s}S, 0, S, 1, s, 0\right]$ from $\alpha_1 = [Ns, 0, s, 1, N, 0]$ for matrix A. The components $\alpha_1[0]$ and $\alpha_1[2]$ allow the computation to access different tiles of matrix A: $\alpha_1[0] = Ns$ allows to go from tiles to tiles of size $s^2 = S$ in the same column, and $\alpha_1[2] = s$ allows to go from tiles to tiles in the same row. These two components become $\alpha_s[0] = \frac{N}{s}S$ and $\alpha_s[0] = S$ respectively. The coefficients $\alpha_1[4] = N$ and $\alpha_1[3] = 1$ allow the computation to access elements in a tile: $\alpha_1[4]$ allows to access element in the same column of the tile, and $\alpha_1[3]$ allows to access elements in the same row of the tile - and stored continuously in memory. They become $\alpha_s[4] = s$ and $\alpha_s[3] = 1$.

The original coefficient that is unitary is left unchanged (e.g., $\alpha_1[3] = 1$ and $\alpha_s[3] = 1$) so a line in memory is a line in cache.

Dynamic mapping, as described for matrix multiply, has a fundamental property: consider for example, α_s and α_1, they define a mapping $\mathcal{I} \rightarrow \mathcal{R}$ so that $A_s = \alpha_s\mathcal{I}$ and $A_1 = \alpha_1\mathcal{I}$ and, by construction, there is a bijective relation between A_s and A_1 ($|A_s| = |A_1|$ and $A_s * \iota_1 = A_s * \iota_2$ if and only if $\iota_1 = \iota_2$ and $\iota_1, \iota_2 \in \mathcal{I}$).

2.2 Fast Fourier Transform

A n-point Fourier Transform, n-***FT***, can be represented as matrix by vector product $\mathbf{y} = F_n * \mathbf{x}$ with $\mathbf{x}, \mathbf{y} \in \mathcal{C}^n$ and $F_n \in \mathcal{C}^{n \times n}$. Each component of $\mathbf{y}$ is the following sum: $y_k = \sum_{i=0}^{n-1} x_i \omega_n^{ik}$, where ω_n is called **twiddle factor**.

When n is the product of two factors p and q (i.e., $n = pq$) we can apply Cooley-Tookey's algorithm. The input vectors $\mathbf{x}$ can be seen as $q \times p$ matrix X stored in row major. The output vector $\mathbf{y}$ can be seen as a matrix Y of size $q \times p$ but stored column major. We can write the n-FT algorithm as follows:

1. for every $i \in [0, p-1]$ we compute $X_{[0,q-1],i} = F_q X_{[0,q-1],i}$ - this is a computation on the columns of matrix X;
2. distribute the twiddle factors $X_{i,j} = \omega_n^{ij} X_{j,i}$;
3. for every $i \in [0, q-1]$ we compute $X_{i,[0,p-1]} = F_p X_{i,[0,p-1]}$;
4. $Y = X^t$.

Algorithms implementing n-FT on $n = 2^\mu$ points are well known, and attractive, because the designer can reduce the number of computations (twiddle factors reductions). However, they are inefficient when the cache has size $S = 2^k$ due to their intrinsic self interference. Implementations, such as FFTW [12], may exploit temporal locality through copying the input data on a temporary work space. Nonetheless, it is difficult to exploit fully spatial locality between the computation of $X_{[0,q-1],i} = F_q X_{[0,q-1],i}$ and $X_{[0,q-1],i+1} = F_q X_{[0,q-1],i+1}$. Two elements in the same column of X will be mapped to the same cache line (self interference), therefore preventing the spatial reuse across the computation of $X_{[0,q-1],i}$ and $X_{[0,q-1],i+1}$. The number of misses due to interference are relatively few, but for large n and in a multilevel memory hierarchy, they are misses at every level - memory pages too. Any improvement in the number of misses at the first level of cache, even small, is very beneficial for a multilevel cache system.

Sub-problems of n-FT access data in non-convex set, therefore our algorithm cannot be applied as it is. We follow a very simple implementation of a recursive algorithm. When we execute the algorithms on the columns of X, the original input matrix X of size $p \times q$ is associated with its twin image in X' of size $p \times (q + \ell)$ where ℓ is the number of matrix elements that can be stored in a cache line. If $q \% \ell \neq 0$, elements in the last and first column of X share the same cache line. If the input vector does not fit the cache, the first and the last column of X are not accessed at the same time; the cache coherence is unaffected.

2.3 Architecture

To describe the effects of the dynamic mapping approach on the architecture design we use the block structure of MIPS R10K microprocessor, Figure 1. We use MIPS but these modifications can be applied to other processors like SPARC64 processors as well. SPARC64 has a large integer register file (RF), 32 registers directly addressable but a total of 56 for register renaming; it is a true 64 bits architecture, and it has only one level of cache.

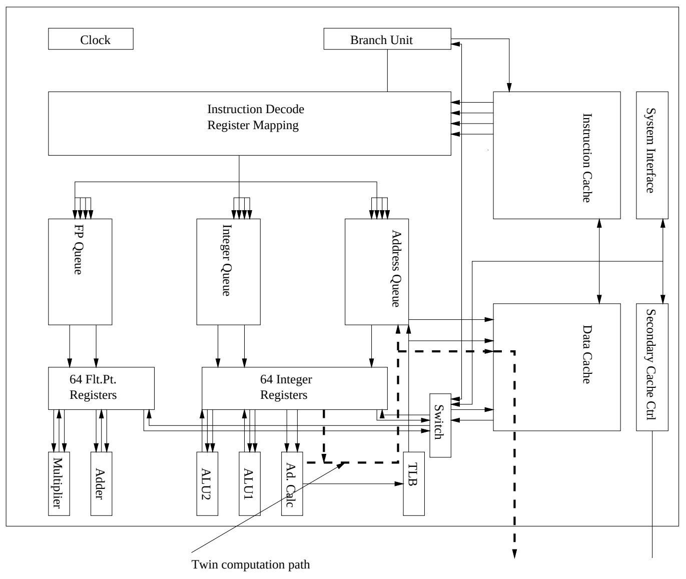


Fig. 1. Proposed architecture, designed based on MIPS R12K

The twin function is performed in parallel with the regular index function. The computations share the same resources, Integer Units and Register file. The twin function result will be stored in the register file, but it is not really an address. The Address Calculation Unit (ACU) and TLB do not process the twin function result. The twin functions need not to be valid addresses in memory at all.

The architecture is as follows. The instruction set is augmented with a new *load* instruction with three operands, or registers: the destination register, the index function register and the twin function register. The load instruction becomes like any other instructions, with two source operands and one destination. To improve performance, a load can be issued as soon as the twin index function is computed to speed up the access in cache. The twin function can be used directly by the cache without further manipulations. The regular index function is really necessary in case of a miss and ACU and TLB must process it. We can imagine that a possible implementation may decide to execute the regular index function only when a miss happens. We can see there is potential to change cache mapping without increasing the hit latency time in cache. The design remains simple: the functional units communicates only with the register files and the register file only with the first level of cache. The design can be applied even when a cache is multi-ported; that is, when multiple loads and writes can be issued to cache.

The proposed approach increases integer register pressure and it issues more integer instructions because of the index computations. The compiler may introduce register spills on the stack, but in general, they are not misses in cache (having temporal and spatial locality). Index function and twin index function are independent and we can issue them in parallel. If the number of pipelined ALUs does not suffice the parallelism available, the index function can lead to a slow down. The slow-down factor is independent from the number of index functions and it is no larger than 2. (The slow-down factor and the total work can be reduced issuing the index function only in case of a miss; we increase only the cache miss latency.)

To avoid consistency problems the data cache is virtually indexed and physically tagged. The new type of load does not affect how many load instructions can be issued or executed per cycle. We assume that an additional Integer RF output port has a negligible effect on the register file access time. (Otherwise, twin functions can be processed in parallel with the index functions and dedicated RF can be used.)

3 Experimental Results

Dynamic Mapping is applied to two applications, Matrix Multiply (Section 3.1, Example 2) and FFT (Section 3.2). We show the potential performance on 5 different systems. Indeed, the algorithms are performed only using the twin function (no regular index function is performed). Here we describe the 5 architectures.

Sun Ultra 5 is based on a Sparc-Ultra-IIi processor 333MHz and Sun Enterprise 250 is based on two Sparc-Ultra-IIe processors 300MHz, both implement 32-bit V8+ instruction set. Their memory hierarchy is: L1 composed of I1=16KB 2-way 32B line and D1=16KB 1-way 32B line 16B sub-block, and L2 is unified off chip U2=1MB 1-way. We have compiled our code with gcc/2.95.3 (g77) and cc (*Sun WorkShop 6* update 1 FORTRAN 95 6.1 2000/09/11). We present the best performance.

Sun Blade 100 is based on a Sparc-Ultra-IIe processor 500MHz. Sun Blade implements 64-bit V9 instruction set. The memory hierarchy is: level L1 composed of I1=16KB 2-way 32B line composed of two block of size 16B and D1=16KB 1-way 32B line 16B sub-block, and L2 is unified U2=256KB 4-way on-chip. We have compiled our code with *gcc* version 2.95.3 20010315 (release) .

Silicon Graphics O2 is based on a MIPS R12K IP32 333MHz, O2 implements 64-bit MIPS IV instruction set. The memory hierarchy is: L1 is composed of I1=32KB 2-way and 64B line and D1=32KB 2-way 32B line and L2 is unified U2=1MB 2-way. We have used f90/cc *MIPSpro Compilers* Version 7.30.

Fujitsu HAL Station 300 is based on SPARC64 100MHz processor, HAL implements a V9 instruction set. Its memory hierarchy is composed on one level of cache with an instruction cache of 128KB 4-way associative and an identical data cache. We have compiled our code with the native compiler *HaL SPARC C/C++/Fortran Compiler* Version 1.0 Feb.

3.1 Matrix Multiply

We implemented the matrix multiply as described in Example 2 in Section 2.1. The application has spatial and temporal locality. Our goal is to show the cache

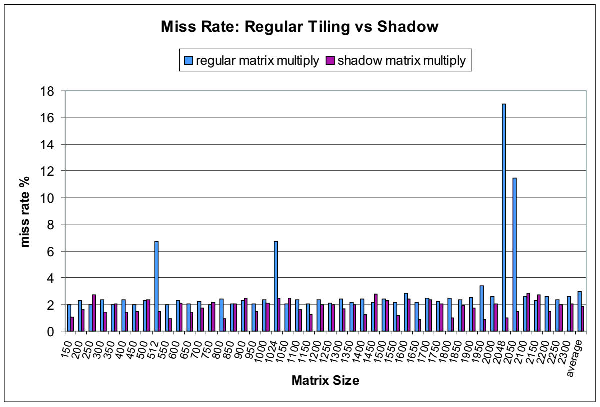


Fig. 2. Matrix Multiply on Blade 100, miss rate comparison.

improvements due to dynamic mapping, Figure 2. We measure the cache performance of matrix multiplication with standard index function and with dynamic mapping. The two algorithms have the same number of operations, memory accesses and most probably the same instruction schedule. The only difference is the access pattern. The dynamic mapping allows a stable miss ratio across different input size (2% which is close to the expected, roughly $\frac{1}{\ell\sqrt{C/3}} = \frac{\sqrt{3}}{2\sqrt{2048}}$) removing cache miss spikes all together.

We always improve cache performance (and therefore execution time). This is expected. We can see a cache miss reduction of 30% in average and up to 8 times cache miss reduction for power of two matrixes.

In this section we do not take in account the effects of register allocation, which can improve cache and overall performance. An optimal register allocation, for tiled and recursive algorithms, can reduce the number of memory accesses and exploits a better data reuse at register level. But at best of our knowledge, there is no known optimal register allocation for matrix multiplication, and no register allocation is able to reduce to zero cache interference for matrix multiplication. Furthermore, register allocation is machine dependent and different libraries use different register allocations (see for examples of register allocation for matrix multiply kernel [7, 31]). We opted to show the cache performance, without the application of other optimizations, in a such a way to quantify the benefits of dynamic mapping alone.

3.2 *n*-FFT

We implemented our variation of Cooley-Tookey algorithm. Our algorithm aims to a balance decomposition of n in factors, such as $n = p * q$ and $p \sim q$. If we can represent the decomposition of a problem in two subproblems as an internal node of a binary tree, the leaves are the **codelets** from FFTW [12] (small FFTs of size between 2 and 16 written as a sequence of straight-line code).

The binary tree can have hight between 1 (n is prime, we need to perform $O(n^2)$ operations) and $O(\log_2 n)$ ($n = 2^k$, we need to perform $(n \log_2 n)$ operations). For some n, a balanced decomposition has a tree of hight $\log_2 \log_2 n$ and it executes $O(n \log_2 \log_2 n)$ operations. The decomposition is static and it is not based on any execution measurements or cost model.

We show the performance of our FFT with and without dynamic mapping in Figure 3 and 4. The bars represent normalized MFLOPS: given an input of size n, the measure reported is $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT. Since our implementation does not have the same number of FLOPS of FFTW, we opted to measure the execution time and determine a normalized number of FLOPS. Every complex point is composed of two float point numbers of 4 byte each (we want to have spatial locality for 16 B cache line size).

We choose to show performance in MFLOPS for two reasons: first, we fixed the leaves computation (codelets) and therefore the register allocation is fixed as well; second, the reduction of cache misses is small, but the performance improvement is extremely significant.

We collected experimental results for four algorithms: we identify with **our FFT**, our implementation of Cooley-Tookey algorithm; **Dynamic Mapping** is **our FFT** when dynamic mapping is applied; we identify with **Upper Bound** a recursive algorithm for FFT that accesses the cache with an ideal pattern (but invalid). When present, **FFTW** is the Fastest Fourier Transform in the West [12]. FFTW is used as reference to understand the relation between the performance of our implementation, the potential performance of dynamic mapping and the performance of a well known FFT. Dynamic mapping could be applied to the FFTW as well, and the improvements would be proportional to the ones shown in the following.

In Figure 3, we show that our implementation is efficient for large power-of-two inputs, but also, it has no steady performance - as FFTW. The performance is a function of the input size. Indeed, the performance is a function of the input decomposition. Large leaves allow fewer computations, therefore better performance. Since the decomposition does not assure that all leaves have the best size, this behavior is expected.

We can notice that for small n, the performance of our implementations may be faster than expected (or slower). This is due to several reasons. One of them is the accuracy problem: the execution of other processes effect the execution-time measure under investigation (e.g., caches, register file, ALU and FPU pipeline trashing). For fairly large to very large problems, where the memory hierarchy is utilized intensely, dynamic mapping lays between its upper bound and our implementation of FFT, as expected.

The characterization of the worst and best performance is twofold: first, it shows the performance we can achieve, the performance we can achieve with dynamic mapping and ideal performance if cache locality is fully exploited; second, when there is no reference for execution time, we are still able to have an estimation of execution time and potential performance.

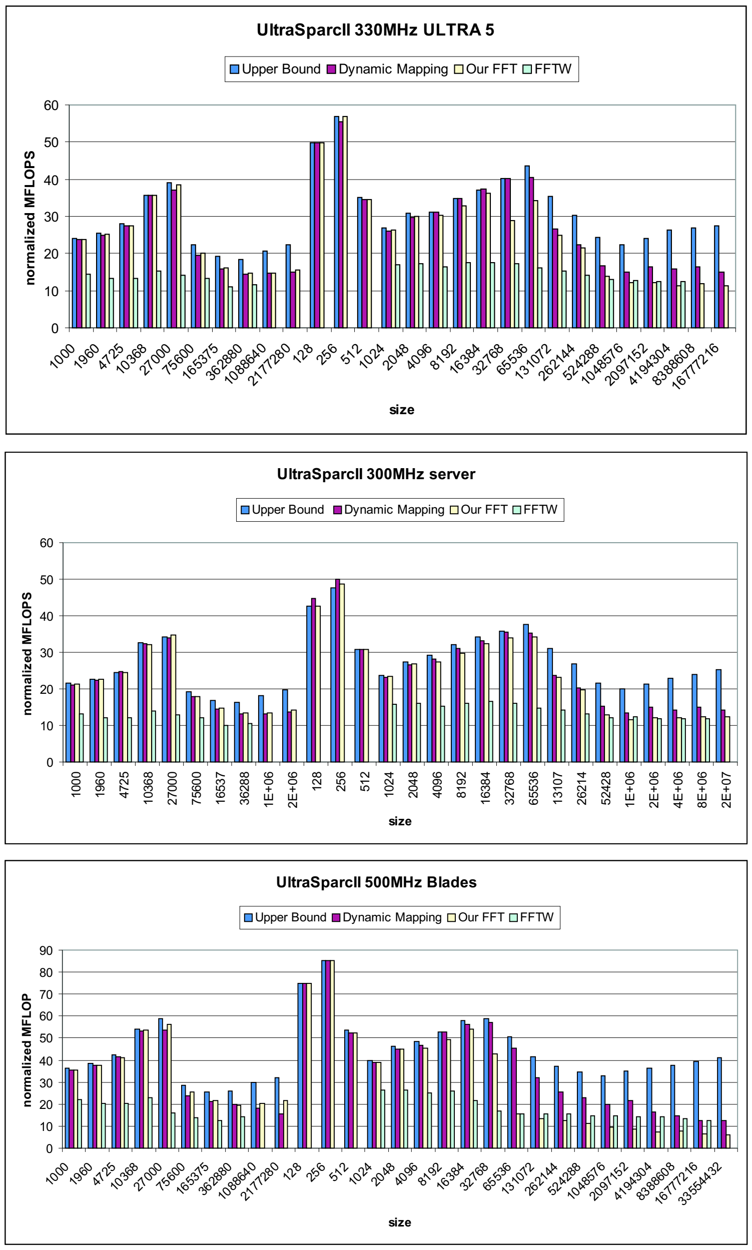
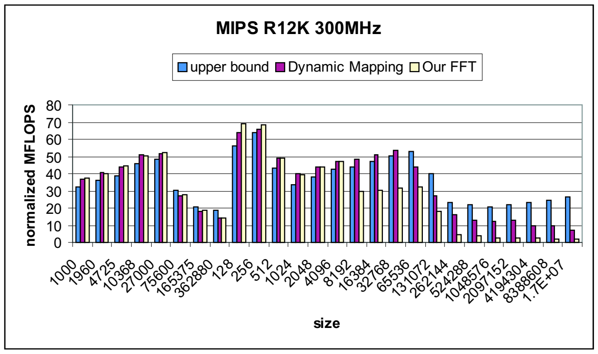


Fig. 3. FFT Ultra 5 (330Mhz), Enterprise 250 (300Mhz) and Blade 100 (500MHz). Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT. The bars from left to right:**Upper Bound**, **Dynamic Mapping**,**our FFT** and **FFTW**



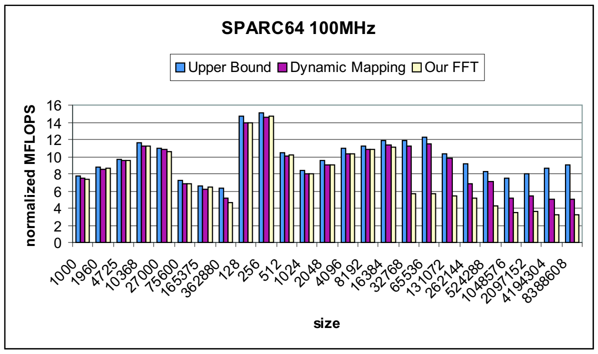


Fig. 4. FFT Silicon Graphics O2 and Fujitsu HAL 300 Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT. The higher the bar the better the performance. The bars from left to right:**Upper Bound**, **Dynamic Mapping**,**our FFT**

FFT is an excellent example of application with high self interference. Even caches with large associativity are affected negatively. In Figure 4, we present performance for two systems with associative caches: Silicon Graphic O2 system and Fujitsu HAL 300.

4 Conclusions

We have presented a software-hardware approach to cope with the loss of performance due to cache interference. Dynamic Mapping exploits spatial and temporal locality, tailoring the cache mapping through a manipulation of the memory addresses. The cache mapping in itself does not change. In general the approach is effective when interference is present, otherwise some slowdown may appear. We present two tests case: matrix multiply and FFT. For matrix multiply we achieve 30% data cache miss reduction and a 8-fold data cache reduction for matrix of size of power of two. For FFT the data cache miss reduction are quantitatively sensitive for large problems and with high interference. We achieve a 4-fold cache miss reduction and such reduction translates into an equal performance speed up.

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers The Foundations.* Kluwer Academic Publishers, 1993.
2. Gianfranco Bilardi and Franco P. Preparata. Processor - time tradeoffs under bounded-speed message propagation: Part II, lower bounds. *Theory of Computing Systems*, 32(5):531–559, 1999.
3. A. Aggarwal A.K. Chandra and M. Snir. Hierarchical memory with block transfer. In *In 28th Annual Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, California, October 1987.
4. A. Aggarwal B. Alpern A.K. Chandra and M. Snir. A model for hierarchical memory. In *Proceedings of 19th Annual ACM Symposium on the Theory of Computing*, pages 305–314, New York, 1987.
5. P. Clauss and B. Meister. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. In *4th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-4*, Toulouse, France, January 2000.
6. M.L.C. Cabeza M.I.G. Clemente and M.L. Rubio. Cachesim: a cache simulator for teaching memory hierarchy behavior. In *Proceedings of the 4th annual Sigcse/Sigue on Innovation and Technology in Computer Science education*, page 181, 1999.
7. G. Bilardi P. D'Alberto and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering 2001*, Aarhus, Denmark, 2001.
8. Paolo D'Alberto. Performance evaluation of data locality exploitation. Technical report.
9. Stephane Eranian David. The making of linux/ia64. Technical report.
10. F. Catthoor N. D. Dutt and C. E. Kozyrakis. How to solve the current memory access and data transfer bottlenecks: At the processor architecture or at the compiler level? In *date*, march 2000.
11. P.R. Panda H. Nakamura N.D. Dutt and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–9, Feb 1999.
12. M. Frigo and S.G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of technology, Sep 1997.
13. Kang Su Gatlin and Larry Carter. Memory hierarchy considerations for fast transpose and bit-reversals. In *HPCA*, pages 33–, 1999.
14. Antonio Gonzlez, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating cache conflict misses through xor-based placement functions. In *Proceedings of the 11th international conference on Supercomputing*, pages 76–83. ACM Press, 1997.
15. R. Gupta. Architectural adaptation in amrm machines. In *Proceedings of IEEE Computer Society Workshop on VLSI 2000*, pages 75–79, Los Alamitos, CA, USA, 2000.
16. J.L. Hennesy and D.A. Patterson. *Computer rchitecture a quantitative approach.* Morgan Kaufman 2-nd edition, 1996.
17. J. Hong and T.H. Kung. I/o complexity, the red-blue pebble game. In *Proceedings of the 13th Ann. ACM Symposium on Theory of Computing*, pages 326–333, Oct 1981.

18. L. Zhang Z. Fang M. Parker B.K. Mathew L. Schaelicke J.B. Carter W.C. Hsieh and S.A. McKee. The impulse memory controller. *IEEE Transactions on Computers, Special Issue on Advances in High Performance Memory Systems*, pages 1117–1132, November 2001.
19. E.D. Granston W. Jalby and O. Teman. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing*, pages 410–419, Nov 1993.
20. Teresa L. Johnson and Wen mei Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
21. T.L. Johnson and W.W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *24th Annual International Symposium on Computer Architecture ISCA'97*, pages 315–326, May 1997.
22. M. Frigo C.E. Leiserson H. Prokop and S. Ramachandran. Cache oblivious algorithms. In *Proceedings 40th Annual Symposium on Foundations of Computer Science*, 1999.
23. W. Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Programming language issues in software systems*, pages 94–6, Orlando, Florida, USA, 1994.
24. J.B. Carter W.C. Hsieh L.B. Stoller M.R. Swanson L. Zhang E.L. Brunvand A. Davis C.C. Kuo R. Kuramkote M.A. Parker L. Schaelicke and T. Tateyama. Impulse: Building a smarter memory controller. In *In the Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 70–79, January 1999.
25. Andre Seznec. A case for two-way skewed-associative caches. In *Proc. 20th Annual Symposium on Computer Architecture*, pages 169–178, June 1993.
26. Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
27. Sharad Malik Somnath Ghosh, Margaret Martonosi. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.
28. P. D'Alberto A. Nicolau A. Veidenbaum and R. Gupta. Static analysis of parameterized loop nests for energy efficient use of data caches. In *Proceeding on Compilers and Operating Systems for Low Power 2001 (COLP'01)*, September 2001.
29. Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.
30. Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, August and September 1994.
31. R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, 1997.
32. M. Wolfe and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 conference on programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991.
33. Zhao Zhang and Xiaodong Zhang. Cache-optimal methods for bit-reversals. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 26. ACM Press, 1999.