Data Cache with Dynamic Mapping

Paolo D'Alberto Alexandru Nicolau Alexander Veidenbaum

University of California, Irvine
School of Information and Computer Science[*]

Abstract

We present an approach to cope with the loss of performance due to cache interference and improve performance predictability of blocked algorithms in modern architectures: e.g. matrix multiply, optimally tiled for a 16KB data cache, achieves 3% data cache miss ratio in average, but it can reach peaks of 16% due to data cache interference. We propose a software-hardware approach, dynamic mapping*, for which the mapping in cache is determined at compile time, by manipulating the address used by the data cache. We present our technique in the context of perfect loop nests and we discuss extension and application to recursive algorithms. We show a speed up more than twofold for matrix multiply (eliminating data cache miss spikes all together and achieving 2% data cache miss ratio in average) and FFT.*

Dynamic mapping *has the same goal of other approaches available in literature, but in our approach the cache mapping is performed/determined while issuing a load, using the computational power of the processor instead of the memory controller. It is an approach combining several concepts like non-standard cache mapping functions and data layout reorganization to reduce data cache interference.*

[*]NSF, Automated Application Development Environment for Future Complex Heterogeneous Computing Platforms. Contacts:{*paolo,nicolau,alexv*}*@ics.uci.edu*

Keywords: cache interference, cache adaptation, blocked algorithms, optimizing compilers, performance evaluation.

Technical Area: Compilers and Languages, Architecture and Algorithms and Applications

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. To avoid CPU stalls due to data/instruction starvation, several approaches have been proposed. We can divide them in three broad and, somewhat arbitrary, classes:

1. Memory hierarchy utilization: the study of performance bounds for families of algorithms, on architectures with memory hierarchy, for example [24, 3, 12, 4];

2. Application reorganization: the study of the trade off between portability and performance across different high performance architectures with multiple level of caches e.g. [2, 32, 30, 5, 19];

3. Adaptive approaches (hardware/software): in hardware few parameters of the architectures, e.g. data cache associativity, may change on demand during the execution of an application, or in software an algorithm may change strategy using less registers to reduce register file power dissipation, e.g. [8, 11].

In this paper, we propose a software/hardware approach that fits into the third category and tries to solve the following problem. Blocked algorithms, like Matrix multiply or FFT, achieve good cache performance in average. Blocked algorithms apply a divide and conquer approach, dividing the problem in smaller problems and try solve them locally. The advantage is clear when the decomposition of the problem exploits the memory hierarchy and the available parallelism, i.e. the data of every subproblem fit the data cache of a processor. The divide and conquer approach improves data reuse and, in practice, the number of communications to/from cache and memory is reduced. But when we measure the cache performance for each input set, we can notice a quite erratic cache behavior, because of cache interference.

For example, an optimal blocked implementation of matrix multiply for an architecture with a directed mapped data cache of size 16KB, can achieve 3% data cache miss rate in average. The average is very close to the optimal cache performance. If we measure the cache performance only for matrices of size power of two, the cache behavior is much different. The data cache miss rate for square matrix of size 2048 is 16%. The difference is due to data cache interference. We propose a compiler/hardware approach to remove data cache miss spikes, changing the cache mapping when needed.

Our approach can be applied to memory references in perfect loop nests, when the memory references are determined as affine function of loop indexes and there is high data cache interference. We show our approach is applicable in conjunction with other optimizations, such as tiling ([21]), and it is designed to exploit spatial locality; in fact, it allows tiling to exploit optimal tile sizes without increasing cache interference, therefore improving reuse and minimizing misses. Furthermore, we show it can also be applied to recursive algorithms. In the scenario we present in this paper and, in general for systems that can be customized to a family of applications, we believe that the utility of loop nest analysis tools, such as the ones proposed by Ghosh et al and D'Alberto et al ([25, 29]), can fully justify and overcome their complexity. These tools can be used to trigger a compiler to apply the optimizations proposed here and achieve higher performance.

We introduce an architecture and a compiler optimization that change the data cache mapping without changing data layout in memory ([9]) and without changing the physical data cache mapping ([23]) (therefore no increase of data cache latency). We achieve this by a one-to-one mapping function, which minimizes data cache interference and ensures reuse. We illustrate the potential performance resulting from our proposed approach for two basic scientific kernels, matrix multiply and FFT. We describe the architectural modifications needed to support the proposed approach. We show that the hardware modifications are modest and require few additional gates, and there is no slow-down of the critical path of cache access. The mapping can change during the execution of an applications, for example on a loop nest by loop nest basis, and it may be different for each loop nest. We assume that the application as very little reuse across different loop nests.

The paper is organized as follows. In Section 2 we introduce the related work. Our software/hardware approach is presented in Section 3. We start with a motivating example in Section 3.1, then we show dynamic mapping while tiling perfect loop nests. In Section 3.2 we apply our approach to recursive FFT algorithms. We present the experimental setup and results in Section 4. In Section 4.3 we propose the architectural modifications to support our approach. Then we conclude in Section 5.

2 Related Work

Caches are predicated on the locality of reference intrinsically available in virtually all programs. Locality of reference is a result of the fact that programs tend to refer to locations in memory that are mostly clustered together either in terms of physical adjacent memory locations (spatial locality) or are referred to one after the other (temporal locality). A memory hierarchy may exploit such characteristics improving average access latency time. A memory hierarchy can be composed of several level of caches, one upon the other (the higher the slower) where the lowest level has an access time of a few cycles. When locality is exploited, it is like data and instructions are always found at the first cache level.

For the application of our approach, we consider simply a two levels memory hierarchy: data cache and memory. Any data going into the cache must be placed to, or evicted from, the memory ([12]). The *allocation* policy, where and how store/read in cache, is based on two basic functions: 1) mapping 2) eviction policy. Briefly, the mapping is a function that given an address in memory is going to return an address in cache. Due to the finite size of caches, several memory locations are mapped on one cache address. For associative caches the mapping returns an address of a set of possible destinations, usually the same cache line in different blocks, or *way*; which way will be allocated is a function of the allocation/eviction policy (in contrast, Skewed caches [23] have a further level of indirection to map an address in each way). Several approaches to achieve almost optimal eviction policy have been proposed and implemented, e.g. Least Recently Used (LRU) (e.g. [7], or for a real architecture example [1]), random etc.

The choice of caches size, cache hierarchy organization, eviction policy, etc, are accomplished as a trade off among complexity of design, die area and average performance over a set of benchmarks. Of course, there are applications for which a configuration could offer better performance than others (e.g. [11]). Unless the architecture is able to adapt to the application needs, the good average performance, which is achieved over a large set of very different applications, may hide large variations in benchmark by benchmark performance. Ad hoc, or *smart*, caches offering dynamic reconfiguration capabilities have been proposed, e.g. of adaptable architectural components [17, 28, 27, 18] (smart caches changes their behavior during the execution, instead Skewed cache the mapping is fixed for the entire execution).

Recently, smart caches have been proposed especially to minimize energy consumption. Indeed, caches are becoming dominant in die area and have high energy consumption. Associative caches are attractive for performance purpose, because they may reduce cache interferences. But they dissipate more energy than direct mapped cache, due to the parallel search of a line in every cache way. Since only one set at any time will contain the data required, a smart investigation of the sets may reduce energy dissipation [16, 14, 15, 26].

Other approaches aim for simpler adaptations, for example line size adaptation. The cache size does not change but, during the execution of an application, the cache line may vary (reducing/increasing the number of lines) so that to reduce misses or to exploit spatial locality. [31, 20, 27]

Embedded systems, like general purpose architectures, may have one or more levels of cache to exploit locality. The main difference is that embedded systems may be *customized*: the software as well as the supporting architecture are tailored for an application and, at times, for a particular input of the application. This opens an exciting opportunity to explore the relation between optimizations considered orthogonal and, often, too time consuming for a general purpose compiler. Layout reorganization such as padding [9] can be revisited and applied in conjunction with other more architectural optimizations, non standard cache mapping i.e. [23] and/or cache replacement policy.

We propose an approach, *Dynamic Mapping*, that 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 is like data layout is reorganized in memory at runtime using the computational power of the processor, but without any data movement. This differs from IMPULSE project [13, 22], which introduces a new memory controller leaving the memory hierarchy untouched. It 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 or to change the memory controller. The cache mapping is defined completely by the application, and it can be driven automatically by a compiler.

3 Dynamic Mapping

In this section we investigate a hardware/software approach to minimize data cache interferences for perfect loop nest with memory references expressed by affine function of the loop indexes. We use a motivating example to introduce the problem and our solution. Then we propose our approach in conjunction with tiling to perfect loop nests, and in Section 3.2, we show that the approach can be successfully applied to a recursive algorithm, FFT.

3.1 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, therefore, it can customize for each reference the data cache mapping.

Example 3.1 *Without loss of generality, 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];
```

For this example we can achieve $2N^2 + 2N^3$ loads/stores instructions. 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 $C[i][j]$ (resp, $B[k][j]$) is $C_0 + N*i + 0*k + 1*j$ (resp, $B_0 + 0*i + N*k + 1*j$).

There is interference at any iteration in the inner loop between $C[i][j]$ and $B[k][j]$ for example when matrix A, B and C are stored continuously and they have size $N = 2^n$ with N grater than the cache size, $S = m \times L$ (L line size and $S = 2^l$), and the cache is direct mapped. In fact, reference $C[i][j]$ and $B[k][j]$ access the same cache line in a direct mapped cache.[1]

We can improve cache performance using the following, twin function to map in cache the memory references: $A_0 + N * i + 1 * k + 0 * j$, $C_0 + N * i + 0 * k + 1 * j + L$ and $B_0 + 0 * i + N * k + 1 * j + 2 * L$. This will assure that $C[i][j]$ and $B[k][j]$ are always in different lines at any loop iteration[2].

Matrix multiply loop nest can be reorganized to exploit temporal locality. We show dynamic mapping is an orthogonal optimization w.r.t tiling [21]. Dynamic mapping is able to reduce self and cross interference independently from tiling, improving further cache performance.

Example 3.2 *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];
```

Tiles of C are read and written N times, B and A are read N and $\frac{N}{s}$ times, respectively. The number of loads and stores are $3N^3(1 + 1/s)$. 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 matrix $N = 2^n > S$ and $S = 2^k$, there is cross interference between references to different tiles (such as $C[i * s + ii][j * s + jj]$ and $B[k * s + kk][j * s + jj]$) and there is self

[1] the relative distance in number of cache lines is: $(C_0+N*i+0*k+1*j-B_0+0*i+N*k+1*j)\mathbf{mod}(2^l) = (2^{2n} + 2^n(i-k))\mathbf{mod}(2^l) = 0$ they are mapped to the same line, $S = 2^l = mL$

[2] $(m2^{2n} + 2^n(i-k) - L)\mathbf{mod}(mL) = -1$, different lines, with $S = 2^l = mL$

interference between any two rows in the same tile (such as $A[i*s+ii][k*s+kk]$ and $A[i*s+ii+1][k*s+kk]$).

When we tile the loop as in Example 3.2, the index function can be described concisely by three vectors, 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 $\gamma_1 * \iota + C_0$.

Every tile is associated with a twin tile, every twin tile is laid down continuously and S elements apart to each other, i.e. elements in different twin tiles will be associated with the same cache portion. The twin function uses the following vectors: $\alpha_s = \left[\lceil \frac{N}{s} \rceil S, 0, S, 1, s, 0\right]$, $\beta_s = \left[0, S, \lceil \frac{N}{s} \rceil S, s, 0, 1\right]$ and $\gamma_s = \left[\lceil \frac{N}{s} \rceil S, 0, 0, s, 1\right]$. The twin functions for A, B and C are $\alpha_s * \iota$, $\beta_s * \iota + S/3$ and $\gamma_s * \iota + 2 * S/3$.

We can see that, e.g., α_s and α_1 describe 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 $A_s * \iota_1 = A_s * \iota_2$ iff $\iota_1 = \iota_2$ and $\iota_1, \iota_2 \in \mathcal{I}$. In fact, we know that we can always lay down the matrices so that the same cache mapping can be achieved). Continuous elements in the original space and in a tile are continuous in the twin space, therefore if there is any spatial locality in the original space there will be in the twin space as well. We have found a twin function, which allows a cache mapping without cache interferences and exploits temporal and spatial locality. The twin function is computationally equivalent to the index function, therefore usual optimizations can apply (e.g. chaining, scheduling, register allocation).

3.2 Fast Fourier Transform

A n-point Fourier Transform, 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 ($n = pq$) and exploiting some property of the twiddle factors we can apply Cooley-Tookey algorithm: the n-point FT is a function of a q-point FT, multiplied by a twiddle factor followed by a p-point FT. If p, or q, is not prime, the problem can be decomposed further. Here, we consider that the q-point and p-point FTs are computed directly, without further decompositions.

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 FT algorithms 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 Discrete Fourier Transform on $N = 2^n$ points are, from the computational point of view, well known and attractive because the designer can reduce the number of computations (twiddle factors reductions). But otherwise, they may be not efficient for a cache based on a cache mapping function $f(x) = x \ \mathbf{mod}\ S$ where $S = 2^k$. Some implementations may exploit temporal locality through copying the input data on a temporary work space [10]. Due to interference, it is very difficult to fully exploit spatial locality available for example 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 in a multilevel memory hierarchy they are misses at every level (memory pages too), therefore very *expensive*. Any improvement in the number of misses at the first level of cache, even small, is very beneficial for a multilevel cache system.

We follow a very simple implementation of a recursive algorithm: the twin function is performed like the input matrix X would be padded on the rows [9]: the original input matrix X of size $p \times q$ is associated with its twin image in X' of size $p \times (q + \ell)$ (for more detail of the code of our FFT see also [6]).

4 Experimental Results

Dynamic Mapping is applied to two applications, Matrix Multiply (Section 4.1) and FFT (Section 4.2). 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). The cache behavior, first level of cache misses, is determined for two architectures by direct measure using registers hardware counters (Ultrasparc IIe and MIPS R12K [1]). Here we describe the 5 different architectures.

- Sun Ultra 5 is based on a Sparc-Ultra-IIi processor 333MHz, Ultra 5 implements 32-bit V8+ instruction set. The memory hierarchy is: L1 composed of I1=16KB 2-way 32B line and D1=16KB 1-way 32B line, and L2 is unified off chip U2=1MB 1-way.

- Sun Enterprise 250 is based on two Sparc-Ultra-IIe processors 300MHz, Sun Enterprise implements 32-bit V8+ instruction set. The memory hierarchy is: L1 composed of I1=16KB 2-way 32B line and D1=16KB 1-way 32B line and L2 is unified off chip U2=1MB 1-way.

  For Sun Ultra 5 and Sun Enterprise 250 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 presented 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.

All systems have a similar Floating Point Units structure and organization. They have a Floating Point register file, a floating point multiply and a floating point adder units with different latency time. For O2 and HAL multiply-add can be pipelined as one instruction. They all have two level of caches, but Fujitsu system. The Sun systems based on UltraSparc processor have similar memory hierarchy but different L2 cache sizes. Data caches and unified caches are direct mapped.

4.1 Matrix Multiply

We implemented the matrix multiply as described in Example 3.2 in Section 3.1. The application has spatial and temporal locality. Our goal is to show the cache improvements due to dynamic mapping, Figure 1. 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

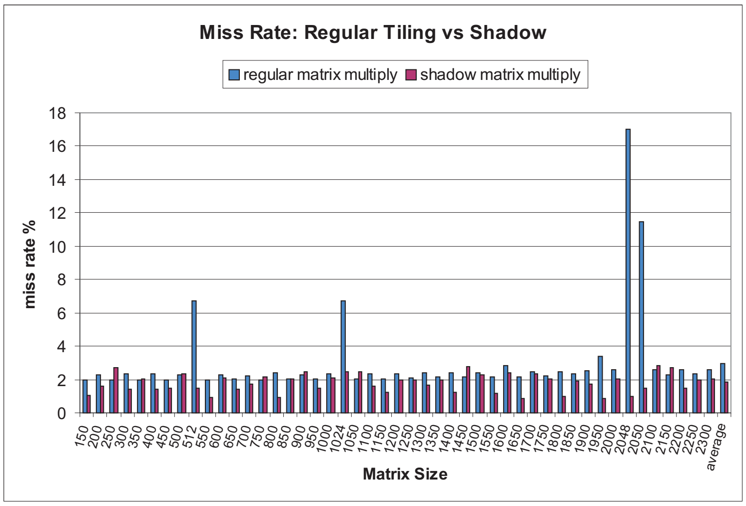

Figure 1: Matrix Multiply on Blade 100, miss rate comparison.

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.

4.2 n-points 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 [10] (small FFTs of size between 16 and 2 written as a sequence of straight-line code). The binary tree can have hight between 1 (n is prime, we need to perform $O(n^2)$ operation) 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 2, 3, 5 and 7. 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).

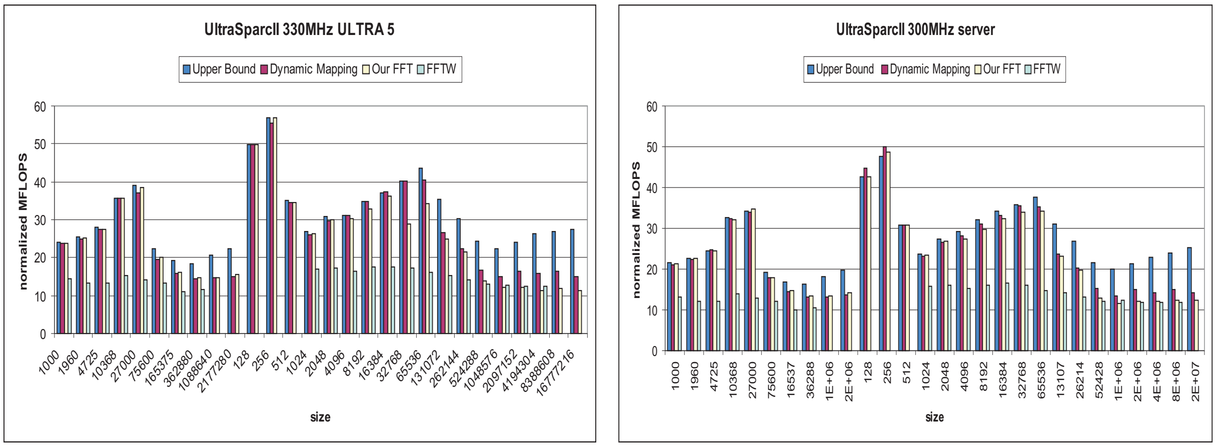

Figure 2: FFT Enterprise 250 (300Mhz) and Ultra 5 (333Mhz). Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT.

We collected experimental results for four algorithms:

1. We identify with *Our FFT*, our implementation of Cooley-Tookey algorithm;

2. We identify simply with *Dynamic Mapping* our FFT when dynamic mapping is applied;

3. we identify with *Upper Bound* a recursive algorithm for FFT that accesses the cache with an ideal pattern (but not valid).

4. When present, FFTW is the Fastest Fourier Transform in the West. FFTW is used as reference, but it is not our goal to compare performance. The comparison is

useful 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.

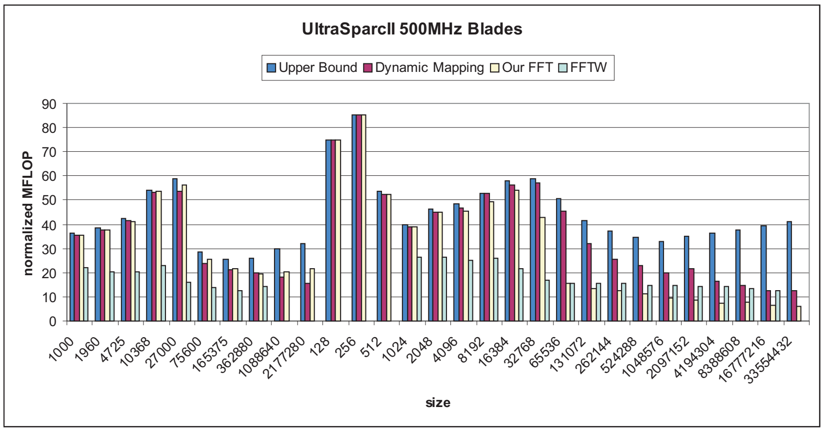

Figure 3: FFT Blade 100. Normalized performance: $n * \log n / (10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT.

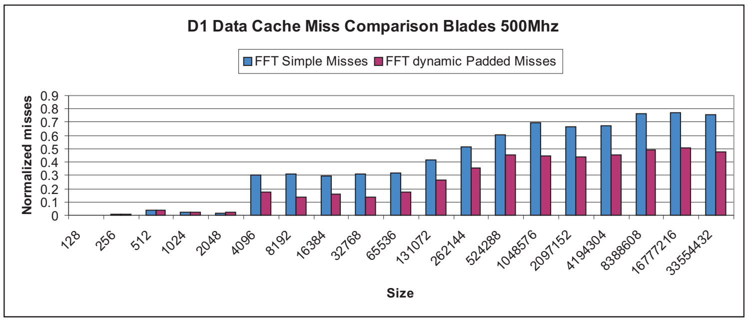

Figure 4: FFT normalized Misses comparison on Blade 100

In Figure 2, we can notice that our implementation is very efficient but also it has no steady performance, that is 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 the best leaves size, this behavior is expected.

We can notice that for small FFTs it may happen that the performance of our implementation may be faster than expected (or slower). This is due to several reasons, one of them is the accuracy problem to measure execution time on a multiprocess system. For fairly large and very large problems, where the memory hierarchy is utilized intensely, dynamic mapping lays between its upper bound and our implementation of FFT.

The characterization of the worst and best performance is twofold: 1) it shows the performance we can achieve, the performance we can achieve with dynamic mapping and ideal performance if cache locality is fully exploited; 2) when there is no reference for execution time, we are still able to have an estimation of execution time and potential performance.

In Figure 3 and 4, we can show three different measures of FFT performance on a Sun Blade 100. Figure 3 shows the normalized performance. Figure 4 shows a comparison between the number of misses of our FFT and the version with dynamic mapping.

FFT is an excellent example with high self interference. Even caches with large associativity are effected negatively. In Figure 5 and 7 we present performance for two systems with associative caches. In Figure 5 we present the normalized performance of Silicon Graphic O2 system. We recall that first level data cache is 2-way associativity and the unified second level is 2 way associative as well. We can notice that performance of our FFT is slightly better than the other algorithms for small size. Indeed, the number of data caches misses is smaller as long as the size of the input is smaller than the first data cache size. In Figure 6 we can see the data cache miss per FFT in function of the input size. In Figure 7 we present the performance of Fujitsu HAL 300. The system has only one level of cache: the data cache is a 4 way associative of size 128 KByte. The dynamic mapping is working and performance is achieved for every input size.

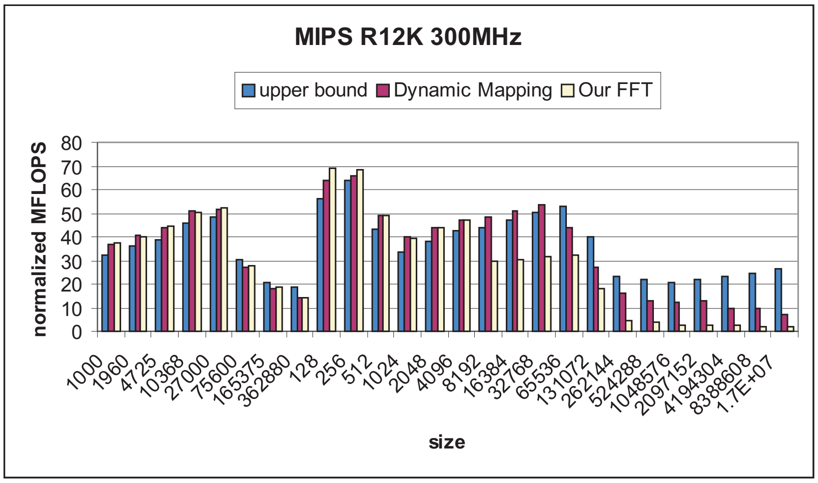

Figure 5: FFT Silicon Graphics O2, Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT.

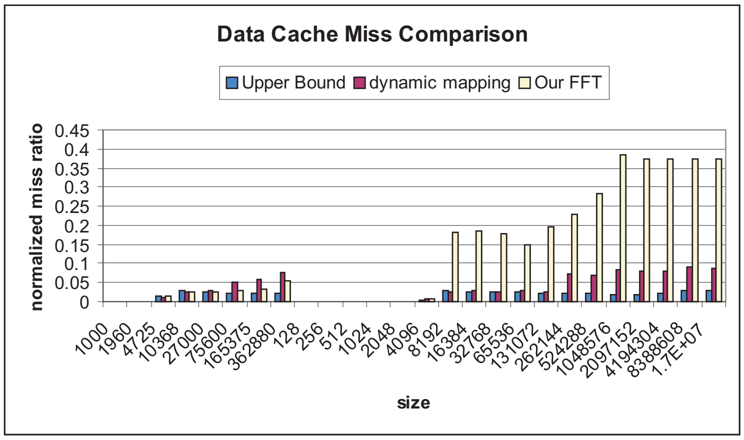

Figure 6: FFT Silicon Graphics O2, Normalized data cache misses.

4.3 Architecture

To describe the impact of the dynamic mapping approach on the architecture design we use the block structure of MIPS R10K microprocessor, Figure 8. 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 result can avoid to be processed by the Address Calculation Unit (ACU) and TLB. The twin function need not be a valid address

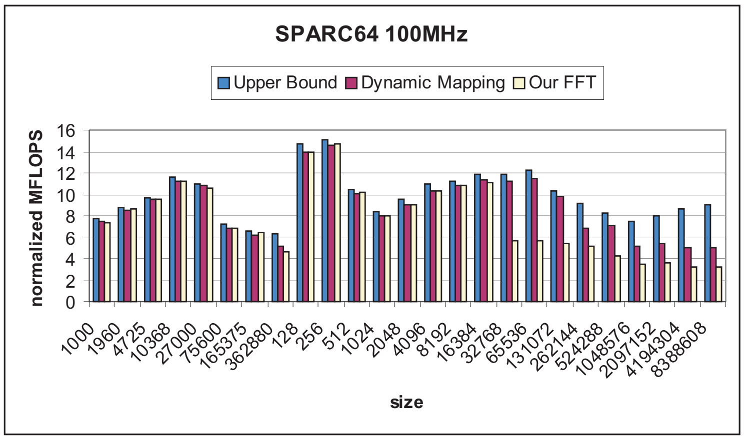

Figure 7: FFT Fujitsu HAL 300, Normalized performance: $n * \log n/(10^6 * timeOneFFT)$, where $timeOneFFT$ is the average running time for one FFT.

in memory at all.

The impact on 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 (but still have available the usual load). The load instruction becomes like any other instructions, with two source operands and one destination. The value of the index function must be processed by the ACU and the TLB must be consulted in case of a cache miss. The twin function can be used directly by the cache without further manipulations. To avoid consistency problems the data cache is virtually indexed and physically tagged.

We assume that such modifications to the load queue will not affect the number and type of instructions that can be issued in parallel at any time. We assume the extra Integer RF output port has a negligible effect on the overall execution time.

5 Conclusions

In this paper we have presented a software/hardware approach to cope with the loss of performance due to cache interference. We propose a software/hardware approach, Dynamic

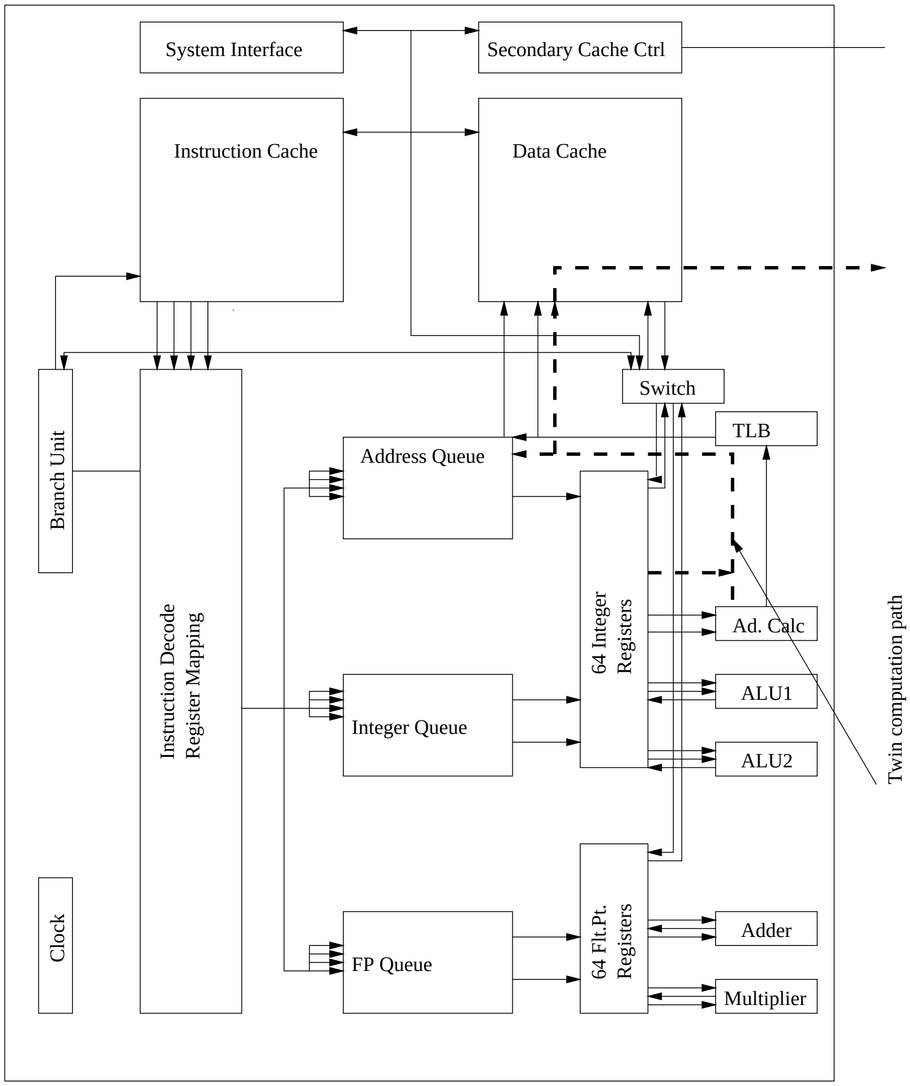

Figure 8: Proposed architecture

Mapping, which exploits spatial and temporal locality customizing the cache mapping through a manipulation of the memory addresses. The cache mapping in itself does not change. A custom mapping function can be achieved by an proper manipulation of the address. In general the approach is effective when interference is present, otherwise some slowdown may appear. We present two tests case: matrix multiply and FFT. Matrix multiply is used to present our approach in conjunction with tiling. FFT is used to present an extension of our approach to recursive algorithms. For matrix multiply we achieve in general 30% data cache miss reduction and up to 8 times data cache reduction for matrix of size of power of two. For FFT the data cache miss reduction are quantitatively sensitive for large problem and with high interference. When the approach is effective the cache miss reduction can be up to 4 times the original number of cache misses. Usually such data cache miss reduction translates in an equal speed up of the performance.

References

[1] *MIPS R10000 Microprocessor. User Manual Version 2.*

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers The Foundations.* Kluwer Academic Publishers, 1993.

[3] 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.

[4] 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.

[5] 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.

[6] Paolo D'Alberto. Zero interferece data cache project. Technical report, UCI, 2002.

[7] A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, pages 143–152, 1990.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J.L. Hennesy and D.A. Patterson. *Computer rchitecture a quantitative approach*. Morgan Kaufman 2-nd edition, 1996.

[13] 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.

[14] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 273–275, 1999.

[15] S. Kim N. Vijaykrishnan M. Kandemir A. Sivasubramaniam M. J. Irwin and E. Geethanjali. Power-aware partitioned cache architectures. In *International Symposium on Low Power Electronics and Design*, pages 64–67, august 2001.

[16] K. Ghose M. B. Kamble. Modeling energy dissipation in low power caches. Technical report, Department of Computer Science. State University of New York, 1998.

[17] A.V. Veidenbaum W. Tang R. Gupta A. Nicolau and X. Ji. Adaptive cache line size to application behavior. In *Proceedings of International Conference on Supercomputing (ICS)*, pages 145–154, June 1999.

[18] L. Benini A. Macii E. Macii M. Poncino and R. Scarsi. Synthesis of low-overhead interfaces for power-efficient communication over wide buses. In *Design Automation Conference*, pages 128–133, june 1999.

[19] M. Frigo C.E. Leiserson H. Prokop and S. Ramachandran. Cache oblivious algorithms. In *Proceedings 40th Annual Symposium on Foundations of Computer Science*, 1999.

[20] S. Manne R. I. Bahar, G. Albera. Power and performance tradeoffs using various caching strategies. In *In International Symposium on Low Power Electronic Design*, pages 64–69, 1998.

[21] M. Lam E. Rothberg and M. Wolfe. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on architectural support for programming languages and operating system*, pages 63–74, Apr 1991.

[22] 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.

[23] Andre Seznec. A case for two-way skewed-associative caches. In *Proc. 20th Annual Symposium on Computer Architecture*, pages 169–178, June 1993.

[24] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[25] 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.

[26] W. Tang, A. Veidenbaum, A.Nicolau, and R. Gupta. Integrated i-cache way predictor and branch target buffer to reduce energy consumption. In *International Symposium on High Performance Computing*, 2002.

[27] W. Tang, A. V. Veidenbaum, and R. Gupta. Architectural adaptation for power and performance. In *International Conference on ASIC*, 2001.

[28] Transmeta. *Crusoe Processor (Model TM5800)*, may 2001.

[29] 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.

[30] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, 1997.

[31] E. Witchel and K. Asanovic. The span cache: Software controlled tag checks and cache line size. In *Workshop on Complexity-Effective Design, ISCA-28*, Goteborg, Sweden, June 2001.

[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.