Proc. Workshop on Challenges in Compiling for Scaleable Parallel Systems at the 8th IEEE Symposium on Parallel and Distributing Processing, October 26, 1996

Tiling for Parallel Execution – Optimizing Node Cache Performance

Wesley K. Kaplow and Boleslaw K. Szymanski
{kaploww,szymansk}@cs.rpi.edu
Department of Computer Science
Rensselaer Polytechnic Institute, Troy, N.Y. 12180-3590, USA

ABSTRACT

Tiling has been used by parallelizing compilers to define fine-grain parallel tasks and to optimize cache performance. In this paper we present a novel compile-time technique, called miss-driven cache simulation, for determining tile size that achieves the highest cache hit-rate. The widening disparity between the processor's peak instruction rate and the main memory access time in modern processor makes this kind of optimization increasingly important for overall program efficiency.

Our technique identifies potential cache misses through compile-time analysis of a loop nest and then processes them on an architecturally accurate cache model. Processing only a small portion of the memory reference trace of a program yields simulation speeds in the millions memory references per second on workstations, with statistics of misses per reference and inter-reference interference counts gathered at the same time.

We discuss the results of applying this method to guide loop tiling for such commonly used computational kernels as matrix multiplication and Jacobi iteration for various cache parameters.

Keywords: Cache, Performance Estimation, Loop Optimization, Tiling.

1 Introduction

Optimization of inter-processor communication, especially for message-passing architectures, have been the traditional focus of automatic parallelizing compilers. However, relatively less attention has been given to the computational performance of each processing node. The importance of the latter has been increasing because current processors achieve high performance only when instructions and data are supplied at a sufficient rate, placing increasing importance on efficient use of cache memory. In many parallel programs, an estimate of the total number of cache lines accessed by the program is essential for predicting the run-time performance [3]. Consequently, compile-time optimizations that improve memory reference locality are relevant to parallelizing compilers.

In scientific programs the various loop nests operating on multi-dimensional arrays are the prime candidates for improvement via compilation optimization. The goal of these optimizations is to change loop nest characteristics to improve memory reference locality. One optimization of this kind, called *tiling* or blocking, transforms a loop nest so that arrays are processed in blocks fitting into the cache, thus eliminating cache capacity misses and reducing cache line interference.

Tiling introduces an additional loop level for each dimension of the arrays that are to be tiled. Its key parameters are the optimal values for the ranges of the added loops, which are dependent on the body of the tiled loop nest and the cache design of the target processor.

1.1 Review of Tiling for Improving Memory Reference Locality

Loop tiling is an optimization technique that has come full circle in its application. Originally explored as a technique to improve the virtual memory performance [1] of uniprocessors, the technique has also been applied to explore fine-grained parallelism exposed by loop skewing and wavefront transformations [11,12] for parallel machines.

Designers of modern multi-processor machines have focused on architectures with the high-speed interconnection of moderate number of fast uniprocessors. The technological improvements both in silicon technology and architectural features have increased the processor speed measured in number of instructions executed per second. However, the access rate of main memory has not kept pace because silicon technology improvements have been mainly used to increase capacity and not speed of memory chips (*e.g.*,use of DRAM and not SRAM for main memory). Cache memory has been used to ease this disparity by placing a relatively small, but high-speed associative memory between the processor and main memory. However, the effectiveness of a processor's data cache[a] is dependent on the data access pattern generated during program execution.

Compilation methods for improving cache performance through modification of a program's memory reference pattern have been the subject of several recent papers [4,8]. Tiling is such a method applicable to loop nests. It requires that the loops are restructured to create the blocked iteration scheme with carefully selected tile sizes. In [8] the authors show that changing tile size can significantly affect the program performance and that the optimum tile size is dependent on both the cache organization and program characteristics.

The focus of this paper is the on a novel compile-time method for accurately determining cache performance characteristics to guide tile size selection process.

1.2 Cache Performance Techniques

Two broad categories of methods to determine cache performance can be identified. The first group, which we call *execution-driven*, includes methods that are accurate, but time-consuming. The second category, referred to as *symbolic*, consist of meth-

[a] Instruction cache effectiveness is not discussed here.

ods that rely on compile time analysis of a program and therefore are less accurate but suitable for inclusion in a compiler. The symbolic methods includes mainly analytic approaches, with the recent addition of compile-time simulation [7].

The execution-driven methods measure the run-time of a compiled program to determine the effect of optimization choices, e.g., tile size selection. The method presented in [5,6] involves capturing memory load and store addresses during execution and processing them via a cache simulation model to determine the miss-rate of a range of parameters. However, such methods involve program execution and therefore are not suitable for embedding within a compilation system.

The symbolic methods include analytic approximations [8,3,2] which estimate the number of cache lines that would be loaded given a semantic analysis of the loop's structure. Their accuracy is limited by the difficulty in accounting for such cache attributes as the line-replacement algorithm, set-associativity, and virtual to physical address mapping. Moreover, these methods cannot accurately link cache miss counts and types with source program components.

In [7], we introduced a novel method called compile-time cache performance analysis. It uses the parse tree of a program to generate a trace of memory accesses that the compiled program would generate if executed on a target. This trace is then fed to an architecturally accurate cache model. The speed of this method is limited by the fact that it is a simulator. However, since we are only interested in the memory addresses generated during loop execution, the computation in the loop nest may be bypassed.

The contribution of this paper is a novel miss-driven cache simulation model in which events are potential cache misses. This model leads to a significantly faster simulation than the ones which processes all array accesses [7]. A cache miss causes an entire line to be read into the cache, so while processing it the simulator can predict what is the next set of indices for the same data structure reference that will access the data beyond the cache line just loaded. Consequently, the simulation bypasses many iterations of the loop nest. An interesting effect of such a miss-driven simulation is that the speed with which program execution is simulated is proportional to the cache miss rate of the simulated loop nest. The simulation's capability to associate cache misses with the source program elements causing them, as well as its ability to model details of cache organization make this approach a valuable compile-time optimization tool.

We discuss the results of applying this method to guide loop tiling for such commonly used computational kernels as matrix multiplication, and Jacobi iteration for such cache parameters as L, the number of bytes per *line* of the cache, K, the number of lines per *set*, and N, the number of sets in the cache.

The rest of this paper is structured as follows. The miss-driven cache simulation method and its algorithms are described in Section 2. Section 3 shows results for both the performance of the simulation method in terms of statistics collected and memory references generated per second, as well as results of using the simulations to guide the tile size determination for loop tiling. Finally, Section 4 contains conclusions and the focus of our continuing research.

2 Miss-Driven Cache Simulation

2.1 Preliminary Definitions

A symbolic reference is a source program item which during execution will generate memory reference. For array elements nested in a loop and subscripted with loop index variables, one symbolic reference will correspond to many memory references, each with different values of the index variables. The reference identifier describes each memory reference in source program terms by augmenting the symbolic reference with the set of loop index values associated with the memory reference. Reference identifiers arranged according to their execution order form the symbolic trace in the same way as the memory references form the program trace. The entire symbolic trace can be generated sequentially, or the simulation can trace only those reference identifiers that represent memory references that may change the state of the cache.

The parser of a source language can be extended to produce an expression that can be used to generate the symbolic trace (see [7]).

The miss-driven cache simulation method processes only the reference identifiers representing potential cache misses. Each time a reference identifier is processed, the simulation determines its effect on the state of the cache model and determines the next candidate miss event. This is achieved by determining the smallest reference identifier, following lexicographically the current one, that is associated with a memory location beyond the content of the cache line holding the current event's memory reference. Such a reference identifier is called the predicted event. Figure 1 shows an example of the cache miss iteration space for a symbolic reference $A[I, 2 * J - 1]$. The event simulation processes only the locations indicated by a circle. Out of a total of 77 memory references, only 18 are processed, a potential simulation speedup of over four times. The actual speedup depends on the specifics of the array subscripts in the simulated reference identifiers and the cache line size used.

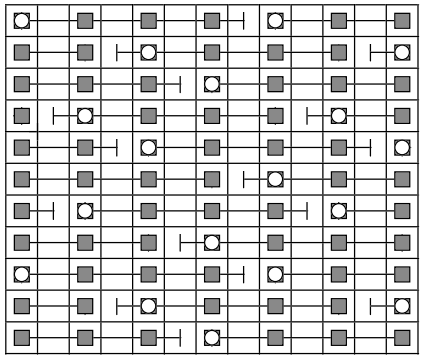

Fig. 1: Example Iteration Space for $A[I, 2 * J - 1]$

A predicted event remains valid if and only if the content of the corresponding

cache line does not change between event creation and processing. To enforce this condition, each line of the simulated cache has a *guard* that indicates if the event is still valid. This guard must only be checked during a cache miss, since this is the only time a cache line can be replaced.

2.2 Simulation Algorithm

The miss-driven simulation algorithm uses an event list and stores the lastly processed event in a so-called *global clock*. There are four main phases to the simulation described in the following subsections.

2.2.1 Creating the Annotated Parse Tree

The first phase creates a standard parse tree and symbol table that are used to determine the initial reference identifiers for the event list, as well as the loop variable limits and base and dimension information for each array. Figure 2 shows an example of the input program.

```
1  A.range[1] = A.range[2] = B.range[1] = B.range[2] = 2048
2  A.base = 10
3  B.base = 200000
4  for k = 1, 1024
5     for i = 2, 256; j = 2, 256
6        A[k,i]=B[k+1,i]+B[i-1,j-1]+B[i+1,j]+B[i,j+1]
6     end
7     for r = 1, 256; s = 1, 256
8        B[r,s]=A[r,s]
9     end
10 end
```

Fig. 2: Sample Simulation Source File

The simulation source language is similar to most imperative programming languages. First, the bounds of each array used must be defined (line 1) which is necessary for calculating the memory offset address for a given reference identifier. Since we are interested in the real (in this case virtual) address of memory reference, we must also define the *base* address of each array (lines 2 and 3) which is added to the array offset address to form the memory location passed to the cache model. The rest of the source code follows the syntax of common loop structured languages.

Standard compiler techniques are used to produce a parse tree, shown in Figure 3 for the code example in Figure 2. Not shown is the symbol table that captures the array range and base information. This tree is the key data structure that is used to create the initial reference identifiers corresponding to symbolic references in the leafs of the parse tree. The tree is also used in other algorithms such as the finding the total number of program memory references at the end of a simulation.

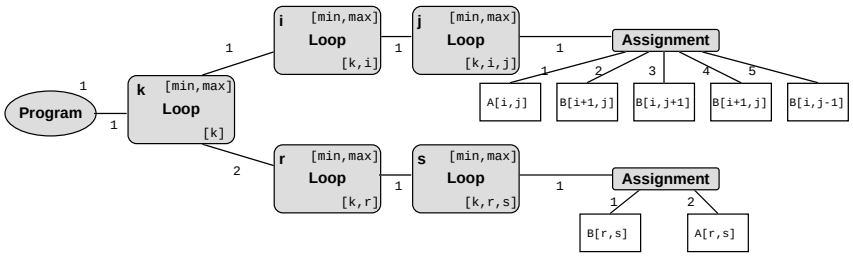

Fig. 3: Example Annotated Parse Tree

2.2.2 Creating the Initial Events and Supporting Data Structures

There are several data structures that must be created and maintained during the simulation such as: initial reference identifiers, next candidate information, loop range data, initial data addresses, ordered event list.

Information from the parse tree is used by the function that computes the next potential cache miss. This includes the $NextCand$ object (Section 2.3), current loop boundaries, and guarded cache model information.

2.2.3 Main Simulation Loop

Figure 4 shows the data flow of the main simulation loop. The basic simulation cycle has the following steps: (i) get the next event, (ii) access the guarded cache model, (iii) perform miss processing, if necessary.

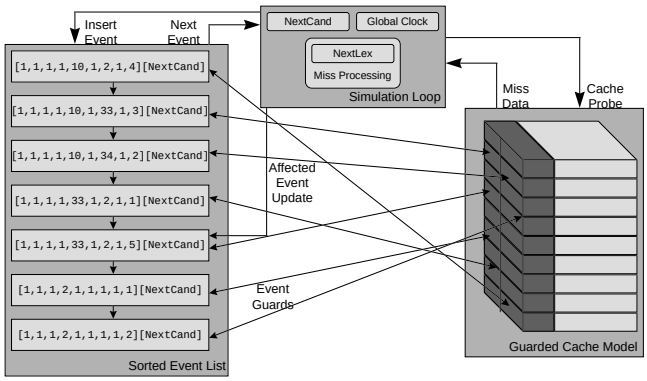

Fig. 4: Event List and Guarded Cache Model

The event list object is a priority queue of events ordered by the lexicographical order of values of loop indexes in reference identifiers of events. In our implementation this is done via an ordered heap. Each event on the event list is guarded by the corresponding cache line in the guarded cache model. A pointer from each event to its guarded cache line is also necessary so that during event processing, the guard can be modified or removed as needed.

The guarded cache object is a conventional model of a cache that can be initialized to support various different cache configurations to accurately model different machine architectures. The current parameters include the associativity, K, the number of sets, N, and the line size, L. For multi-way caches ($K > 1$) the model implements the least-recently-used (LRU) algorithm for determining which cache line to replace.

Figure 5 shows the main event loop code for the simulation. In each cycle, the minimum event from the event list is the next reference identifier to process as by definition of event list all preceding reference identifiers have already been processed. The event is taken off the event list and its cache line guard is removed. Finally, the global clock is set to this event.

The reference identifier for this event is then used both as an input to the cache model, and to the $NextCand :: getnext$ procedure that predicts the next candidate miss for the relevant symbolic reference. The *getnext* procedure, described later, requires the array offset of the previous event to compute the next candidate event, hence this information is stored in the event structure.

The cache model is accessed and the action of the cache is recorded, including the cache line that was accessed and the type of access (hit or miss). The current event is updated to contain the reference identifier of the predicted candidate miss using the *getnext* procedure. The event is then set to be guarded by the saved cache line, and the cache line is set to guard the event. The updated event is then placed back on the event list.

If the action of the cache was a hit, or if the action was a miss and the accessed cache line did not have an active guard, then no further processing is required. However, if the action was a miss and the accessed cache line was guarding an event, then that event is no longer valid because it refers to a cache line replaced by the currently processed miss. The guarded event is rolled back to the first reference identifier with the symbolic reference corresponding to the replaced cache line that is larger than the global clock. The procedure $Event :: nextlex$ performs the required index update for the affected event.

The *nextlex* procedure updates the reference identifier of the affected event and the address for this event is then calculated (for subsequent use when this event is processed). Finally, the event is re-inserted into the event list and marked as unguarded. The simulation cycle then repeats.

2.3 Next Miss Candidate Prediction Algorithm

The prediction of the candidate cache misses for a symbolic reference is made by the $NextCand$ algorithm.

The $NextCand$ algorithm relies on a data structure that is dependent on the syntax and semantics of an array reference. There is one data object per symbolic reference in the source program which contains the coefficients necessary to characterize each affine subscript.

From this information, it is possible to determine the values of the *prod* array. The values of this array represent the number of memory locations that are moved

```
Event*  Eptr, gEptr
RefVector GlobalClock
while ¬done {
  Eptr = EventList.top()GlobalClock = *Eptr
  if Eptr->getguard() = GUARDED
    CacheModel.setguard(Eptr->getway(), Eptr->getset(), NULL)
  CacheRec probe
  int address = Eptr->address
  int action = CacheModel.Probe(address, probe)
  Eptr->address = Eptr->getnext(*Eptr,
      Eptr->min, Eptr->max, address - Eptr->base, cachelinesize)
  CacheModel.setguard(probe.getway(), probe->getset(), Eptr)
  Eptr->setway(probe.getway())
  Eptr->setset(probe.getset())
  Eptr->setguard(GUARDED)
  EventList.insert(Eptr)
  if action = MISS {
    gEptr = probe.getguard()
    if gEptr¬NULL {
      gEptr->setguard(¬GUARDED)
      Eptr->nextlex(GlobalClock)
      gEptr->setway(probe.getway())
      gEptr->setset(probe.getset())
      EventList.insert(gEptr)
    }
  }
}
```

Fig. 5: Miss-Driven Cache Simulation

for each increment of each respective index. This information is used to calculate the offset of address of an array reference. It is also used in the $NextCand :: getnext$ function to determine the set of index variables for this symbolic reference that will result in memory reference beyond the current cache line (this set defines the next candidate miss for this reference).

Omitted here, due to length constraints, is the detailed description of the *getnext* routine. The essence of the algorithm is illustrated in Figure 6 in which the starting index variables are $[I = 5, J = 3]$. First, based on the stored address, the offset inside of the current cache line is determined. Next, starting from the innermost towards the outermost index variable (defined by array I) the minimum increment, inc, to move outside the cache line is determined by dividing the number of bytes needed by the *prod* of the current index. This is then used to create provisional values for the next reference's index set, $prov$. In this case of upper box in Figure 6, the provisional reference does not exceed the maximum loop ranges, so the address for this reference is calculated for use in the next call and the provisional value is

the new current (predicted) reference identifier.

The situation is a bit different in the example in the lower box. Here, the increment causes *prov* to move outside of the valid index space. The algorithm then increments the next higher index variable, and the lower index is reset to its minimum value. The algorithm determines that it must move further because we have not moved outside of the cache line. A new increment is determined for the lower index and applied to create a new provisional value. This is again checked against the maximum ranges, and the provisional is made the new current index set, and finally the address is calculated.

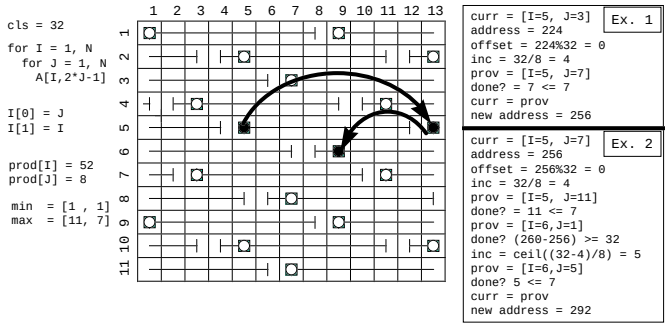

Fig. 6: Next Candidate Miss Examples

2.4 Next Lexicographical Reference Algorithm

When a cache line is replaced due to a cache miss, the event guarded by the cache line is no longer valid. In this case we must roll back the affected event. Since all reference identifiers before the global clock have been processed, we need the least set of values for the indices of the affected event's symbolic reference that follow the global clock. The algorithm in Figure 7 performs this function.

The algorithm works by copying all of the index information from the outermost loop towards the innermost loop as long at the symbolic reference corresponding to the global clock and to the affected reference identifier are the same. Once the loop nests are different, the remaining indices in the affected reference identifier are set to the minimum for each index respectively.

2.5 Statistics

2.5.1 Total Number of References

Although we know exactly the number of events processed, we do not know how many references we executed in total because each event represents a varying number of references.

The number of actual references is determined by summing the number of executions for each symbolic reference. To determines this number, the algorithm compares the reference identifier with the global clock. This comparison detects

```
Event :: nextlex(RefVector& after)
{
  istack* thislive, afterlive
  thislive = this->treeptr->getlive()
  afterlive = after.treeptr->getlive()
  int minlen = thislen = this->last
  afterlen = min(afterlen, minlen)
  for( int i; i < minlen; i++)
    if thislive[i] = afterlive[i]) curr[i] = after.curr[i]
    else for( int r = i; r < thislen - 1; r++){
      curr[r] = min.curr[r]
      break
    }
  if thislen = afterlen & r = thislen& *this < after {
    last()++
    normalize(min, max)
  }
}
```

Fig. 7: Next Lexicographical Reference Algorithm

whether or not the symbolic reference was executed in the last iteration of the loop nest. This is done by moving down the parse tree and comparing the statement selectors of the global clock and the desired reference. There are two cases:
1. The reference identifier is no greater lexicographically than the global clock. Here, the symbolic reference must have executed as many times as its enclosing loops, up to the point that the global clock and the reference identifier are the same, and for the full range of all of the loops between the symbolic reference and the above point.
2. The reference identifier is larger than the global clock. In this case the symbolic reference has been executed as many times as the loops it shares with the global clock.

2.5.2 Cache Statistics

The unique feature of the presented simulation method is its ability to associate gathered cache miss statistics with the symbolic references. The statistics currently gathered are:

Total Cache Miss Rate. This is the total count of all miss accesses to the cache.

Per Symbolic Reference Miss Rate. The number of times an event with this symbolic reference causes a miss.

Per Reference Interference. The number of times a symbolic reference interfered with another.

Per Array Intrinsic Miss Rate. The number of times a memory reference to an array causes a miss but does not displace a cache line containing elements from an array in the program.

Per Array Cache Interference Rate. This includes both self-interference and cross-interference. Each cache line in the guarded cache model is tagged with the array to which this line belongs. During a miss the tag value is compared with the identity of the array accessed by the event's reference.

Figure 8 shows a sample output of the event simulation. The numbers to the right of each symbolic reference indicate the number of times this reference interfered with the other symbolic references. Figure 9 shows an example graph of miss rate components for various tile sizes.

```
Cache Parameters:  K = 1 L = 4 N = 5
Num.  of Refs.:  40208  Num.  of Misses:  27958  Misses per event:  0.931933
Misses per ref.:  0.695334 Intrinsic Miss rate:0.000795862
Interference Data:
 A B C
A 0 2040 0
B 2013 23873 0
C 0 0 0
Group Self Interference rate:  0.593738  Inter-Group Interference rate:  0.100801
      i.range = 2,127; j.range = 2,127
      (2042) 7.30381 A[i,j] [# 0] 0 0 0 0 0
      (5937) 21.2354 B[i,j-1] [# 1] 0 0 0 3894 0
      (7979) 28.5392 B[i-1,j] [# 2] 0 6000 0 0 1979
      (7979) 28.5392 B[i+1,j] [# 3] 0 0 7979 0 0
      (4021) 14.3823 B[i,j+1] [# 4] 0 0 0 2042 0
```

Fig. 8: Cache Statistics and Annotated Source Listing

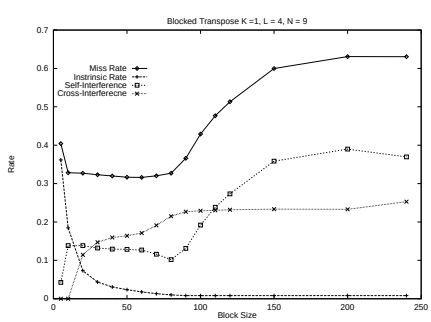

Fig. 9: Example Graph of Miss Rate Components

3 Results

In this section we present some preliminary results of the compile-time miss-driven simulation method. The results are intended to demonstrate that the method can quickly generate the cache performance statistics required to select the tile size at compile-time. They also show that the additional fidelity of this method, as compared to analytical techniques, provides valuable insight for the selection of optimum tile sizes.

3.1 Simulation Performance

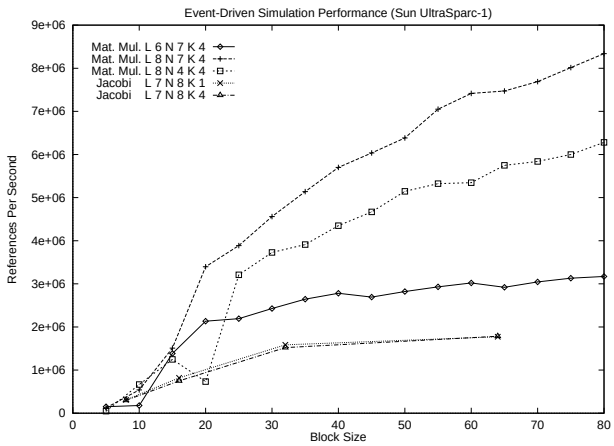

Fig. 10: Reference Simulation Rate for Matrix Multiplication

Figure 10 shows the performance of our simulation method on two different loop nests: matrix multiplication and Jacobi Iteration. The performance is plotted against block size for the different loop nests and cache parameters, and represents the number of references simulated per second. These numbers compare favorably to real target execution in terms of instructions per seconds because execution of real code on a target includes also execution of loop control statements and calculations of the array offsets and values. The worst performance was achieved with small tile sizes because of the high miss-rate experienced.

As described in Section 2.3, the cache prediction is based on reference self-spatial locality and therefore the performance is proportional to the simulated cache line size. However, the performance is also inversely proportional to the miss-rate, as the larger miss-rate generally implies more cache interferences and therefore generates more event roll-backs due to invalidated events. This accounts for the dip (block size = 20) in the matrix multiplication case with $L = 8, N = 4, K = 4$.

On a Sun UltraSparc-1, the generation of the matrix multiplication tile size graph takes approximately 1 second in the current implementation. The current simulation code is written in C++ and its object-oriented design is intended for ease of implementation and adaptation and not performance. Changes to the code,

such as making some of the simple variables *public* in outside of their class have already yielded significant performance improvements.

3.2 Compile-time selection of Tile Sizes

The accuracy and speed of the cache model can be used to quickly determine and optimal tile size for a problem. The left graph in Figure 11 presents the cache performance for various tile sizes for two different matrix multiply problem sizes. As shown, for the same cache architecture and algorithm the tile size can depend on small changes in problem size (as shown in [8]). The right graph in Figure 11 gives miss-rates for matrix-multiply on a $32K$ cache for three different cache organizations. The graphs show that there is a significant dependence on the structure of the cache, and that cache performance analysis methods that rely on the total number of cache lines may yield inaccurate results.

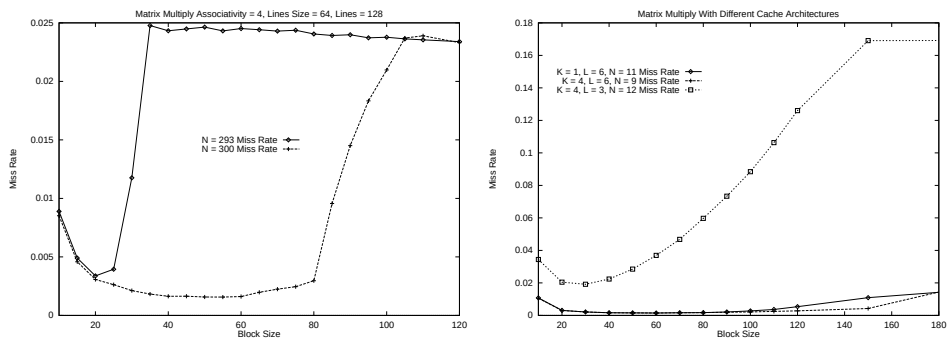

Fig. 11: Matrix Multiplication for various 32K Cache Architectures

Figure 12 shows another example of a loop nest, adapted from [10] and representing a typical stencil computations. As with the matrix-multiplication, we can see from Figure 13 that optimum tile size for the same size cache depends on the cache's structure.

```
double A[N,N], B[N,N]. C[N,N]
  for ii = 1, N,T
    for jj = 1, N,T
      for i = i, min(ii+T,N)
        for j = 1, min(jj+T,N)
          A[i,j]=A[i+1,j]+B[i,j]+B[i,j+1]+C[j,i]+C[j,i+1]
        end
      end
    end
  end
  ...
```

Fig. 12: Tiled Stencil Code

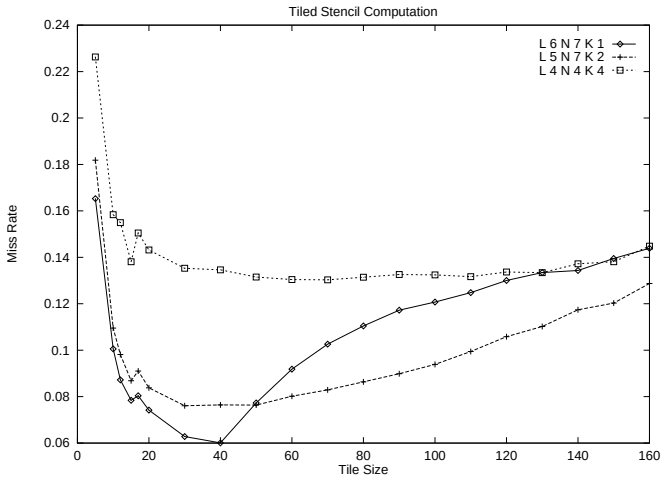

Fig. 13: Results for Tiled Stencil Kernel for Various Cache Parameters

Figures 14 and 14 show the correlation between the simulated performance of the tiled matrix transpose and the real target performance, demonstrating the fidelity of the cache statistics in predicting the real program performance.

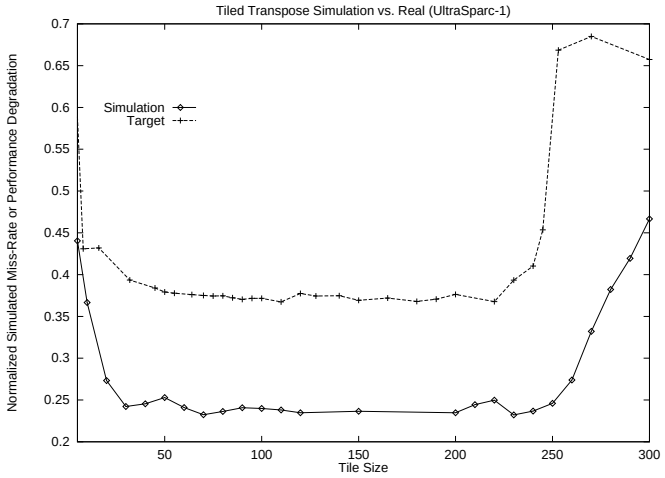

Fig. 14: Performance of Tiled Matrix Transpose

4 Conclusions

We have developed and described a novel compile-time method for determining the cache performance of loop nests to assist in the selection of optimal tile sizes that improve dense loop performance. The miss-driven method provides accurate cache performance thanks to the use of a real cache model instead of an analytic approximation. Moreover, the cache model and miss-driven structure of the simulation

provides detailed information about the nature of the cache misses. Each symbolic reference is assigned the number of times this references cause a miss, as well as the number of times this reference displaced others in the cache. This information can be used to direct other loop optimizations such as loop permutation, fusion, and distribution [9]. Moreover, per reference based information is critical to whole program cache optimization [10].

Acknowledgments

The authors would like to thank their colleagues from Rensselaer Polytechnic Institute: Peter Tannenbaum for his help in implementing the parser and Charles Norton for help in the C++ implementation of the simulator. This work was supported in part by Lucent Technologies, Bell Laboratories and by NSF Grants CCR-9527151 and CCR-9216053. The content does not necessarily reflect the position or policy of the U.S. Government.

References

1. W. Abu-Safah, D. J. Kuck, and D. H. Lawrie. Automatic program transformations for virtual memory computers. In *Proceedings of the 1979 National Computer Conference*, pages 969–974, June 1979.
2. S. Carr, K. McKinley, and C W. Tseng. Compiler optimizations for improving data locality. In *ACM Architectural Support for Programming Languages and Operating Systems, San Jose, CA*, October 1994.
3. T. Fahringer. Automatic Cache Performance Prediction in a Parallelizing Compiler. In *Proceeding of AICA 1993, Lecce/Italy*, September 1993.
4. D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation . *Journal of Parallel and Distributed Computing*, October 1988.
5. A. Goldberg and J. Hennessy. Performance debugging shared-memory multiprocessor programs with mtool. In *Processings of Supercomputing 91*, 1991.
6. A. Gupta, M. Martonosi, and T. Anderson. Memspy: Analyzing memory system bottlencks in programs. *Performance Analysis Review*, 20(1), 1992.
7. W. K. Kaplow and B. K. Szymanski. Program optimization based on compile-time cache performance prediction. *Parallel Processing Letters*, 6(1):173–184, 1996.
8. M. S. Lam, E. E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. ACM ASPLOS, Santa Clara, CA*, pages 63–74. ACM, NY, April 1991.
9. Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Lanaguages and Systems*, 18(4):424–453, July 1996.
10. Kathryn S. McKinley and Oliver Temam. A quantatative analysis of loop nest locality. In *ASPLOS-VII*. ACM, 1996.
11. Balram Sinharoy and Boleslaw Szymanski. Finding optimum wavefront of parallel computation. *Journal of Parallel Algorithms and Applications*, 2(1):5–26, 1994.
12. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel and Distributed Systems*, 3(10):452–471, 1991.