Proc. 28th Hawaii Int. Conference of System Sciences, Maui, HI, January, 1995, IEEE Computer Society Press,
Los Alamitos, CA, 1995, vol. II, pp. 93--102

Impact of Memory Hierarchy on Program Partitioning and Scheduling *

Wesley K. Kaplow†, William A. Maniatty, and Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute, Troy, N.Y. 12180-3590, USA

Abstract

In this paper we present a method for determining the cache performance of the loop nests in a program. The cache-miss data are produced by simulating the loop nest execution on an architecturally parameterized cache simulator. We show that the cache-miss rates are highly non-linear with respect to the ranges of the loops, and correlate well with the performance of the loop nests on actual target machines.

The cache-miss ratio is used to guide program optimizations such as loop interchange and iteration-space blocking. It can also be used to provide an estimate for the runtime of a program. Both applications are important in scheduling programs for parallel execution. Presented here are examples of program optimization for several popular processors, such as the IBM 9076 SP1, the SuperSPARC, and the Intel i860.

1 Introduction

1.1 Background and importance

The recent advances in processor technology, both in execution efficiency in terms of cycles-per-instruction and in raw clock rate, have dramatically improved single processor performance. These processors are also being incorporated into the next generation of Massively Parallel Processors (MPPs). However, these processors perform at full speed only when programs exhibit enough locality for data caches to sustain high hit-rates. When such locality does not exist, the performance drops from the CPU's cycles-per-instruction speed to the much slower memory-reference access speed.

Cache performance is therefore an important consideration in the development of compiler code optimization techniques. Given a description of a target computer and a program, the compiler should be able to optimize the generated code to take advantage of the cache architecture.

In this paper we develop a *dynamic* cache performance estimation technique that determines the miss-rate of a section of an application code using an architecturally accurate cache simulator. The miss-rate information can be used to guide applicability of compilation optimization techniques, as well as to assist in providing a more accurate execution cost estimate necessary for scheduling algorithms (c.f.,[12, 5]).

1.2 Applications of cache performance predictions

For many scientific numerical computations a large portion of the execution time is spent in *nested loop* structures that iterate over several, generally multi-dimensional, arrays, performing *stencil computations*. These are computations in which the evaluation of each new data element requires the data from some non-data-dependent neighborhood in one or several arrays. Unfortunately, the performance of the stencil computations are often highly non-linear as a function of the iteration ranges of the enclosing loop nest. Figure 1 illustrates this effect for an IBM SP1 processing node executing the Jacobi iteration in a solver for partial differential equations. The machine performance is plotted as a function of the size of a square array and

*This work was supported in part by ONR grant N00014-93-1-0076, by NSF grant CCR-9216053 and by a grant from the IBM and AT&T Corporations. The content of this paper does not necessarily reflect the position or policy of the U.S. Government—no official endorsement should be inferred or implied.

†{kaploww,maniattb,szymansk}@cs.rpi.edu

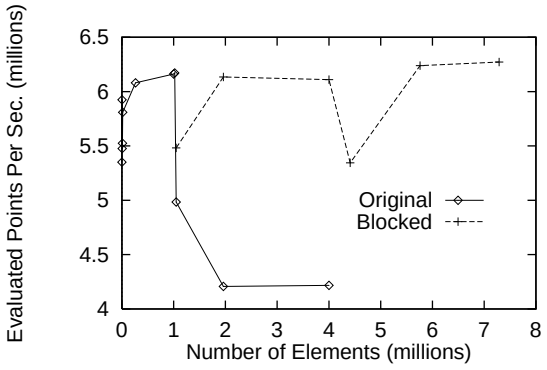

Figure 1: SP1 Single Processor Performance on Jacobi Iteration

is measured in millions of points computed per second. The figure shows that the performance of the code is highly dependent on the size of the array computed and it drops by more than 25% at the array size of approximately 1024^2. Since the computational complexity remains constant, this is caused by a decrease in cache effectiveness.

Loop nests are prime prospects for code optimization. Many loop restructuring optimizations influence cache performance. Examples are loop interchange, fusion, distribution, iteration-space blocking, and skewing (c.f., [15, 9, 4]) which can dramatically improve the performance of loops and therefore programs (c.f., [8, 11]). For example, Figure 1 shows the application of an *iteration-space blocking* optimization, which can improve performance by preserving the locality of reference independently of problem size. For large problem sizes, blocking modifies the loops in such a way that arrays are iterated over in blocks whose size is less than the cache performance degradation point.

Central to the use of loop optimizations is the development of a metric that can be used to determine the relative cache performance for an original and optimized loop nest. For other compiler optimizations, prediction of the cache performance degradation point is of interest as well.

1.3 Methods for cache performance prediction

Methods proposed for determining the cache performance of a loop nest can be classified into two groups. *Dynamic* techniques require program benchmarking with real or synthetic address traces. *Static* techniques attempt to estimate cache performance based on an analysis of program components such as array accesses contained within the loop nest.

Program benchmarking involves compiling and executing code segments on the target computer. The cache effectiveness can then be estimated by the deviations of the execution time from the algorithmic complexity of the segment.

The effectiveness of a cache can also be measured by extracting the memory reference *trace* of a program and then running the trace through an architectural simulator. There are both hardware and software methods for capturing the reference traces. Once obtained, the they can be fed into an architectural simulator of the cache. The greatest drawback to this approach is that to be effective, traces have to be millions of references long [13]. Another problem, especially relevant to scientific numerical codes, is that the identity of the program components and structure that generated the address trace is lost, and therefore it is difficult to make conclusions about how to modify the source code to improve performance.

Static techniques use program analysis to estimate the number of cache misses generated by a program fragment. Most of the recent work has focused, as we do, on the loop nests of a program. Porterfield [11] estimates the number of cache lines referenced by a loop, but considers only caches with unit line size. Moreover, his method is applicable only to loops with constant data dependency vectors. Ferrante et al.[7] determine an upper bound for the number of distinct cache lines referenced in a program using a detailed analysis of the data dependency of array references and index expressions in loop nests. Fahringer[3] develops the notion of *array access classes* that are created by grouping together array references that exhibit the same spatial and temporal reuse. He applies this method successfully to *loop interchange* and *loop distribution* optimizations. Wolf and Lam [9] develop a cache model based on the number of loops carrying reuse. They classify cache misses into two groups: cross interference, which is the interference between two different variables, and self interference, caused by references to the same array. They develop a procedure to determine the blocking parameters to eliminate self interference, which they found to cause the largest increase in miss-rates. Temam et al.[14], attempt to derive analytical expressions for the cache miss rates. They carefully examine the role of cache interferences, but their analysis is limited to direct-mapped caches and rectangular loops with only one loop index per dimension.

All the above methods analyze a program loop nest to determine an expression that represents the number of cache misses for that nest. Some of the methods

are limited by the types of loop nests that can be analyzed, others by the cache types that the model can represent. Their main advantage in comparison to dynamic methods discussed earlier is speed, their main drawback is accuracy.

Several researchers evaluate the performance of program code sections using linear models in which the estimated time is equal to the execution frequency times the cost of each operation. This cost is not adjusted for memory hierarchy effects. A performance analysis methodology given in Clement et al., [2] uses a linear model to determine program execution time with the additional consideration that all variables that are shared between processors are not cached and have a larger access time than local variables. Gerasoulis and Yang [6] estimate the execution cost of nodes in a task graph by the number of arithmetic operations and communication operations and the time required per operation on a target. Balasundaram, et. al [1] use a *training-set* method to determine the cost of elementary operations on a target and then use a linear model to determine the cost of a section of code.

1.4 Organization of the paper

In this paper we present a simulation based approach that can be used with a class of loops that are characteristic of a majority of scientific programs.

Figure 2 shows how our method integrates with the static loop cost estimate to produce a performance prediction. The performance prediction is then used to guide loop optimizations. The shaded portions of the figure show the extent of this paper.

The remainder of this paper is structured as follows. Section 2 describes the program structure and an architectural model of the processor used in simulations. Section 3 presents the architectural cache simulator. Section 4 validates the simulator through experiments with several common benchmark programs and machine architectures. In section 5, an heuristic method is shown that reduces the execution time required to precisely determine the performance degradation point for a program fragment and therefore makes the overall method practical for use in a compilation system. Conclusions and goals for further research are given in Section 6.

2 Architectural and program model

This section presents the cache architectural model used in the simulation, and the application program model used to drive the simulation.

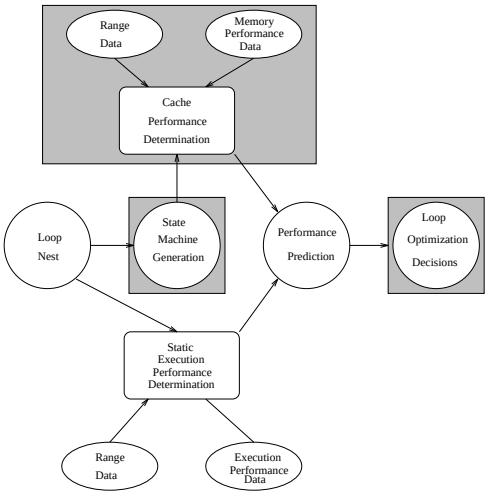

Figure 2: Execution Cost Determination for a Loop Nest

2.1 Architectural model and parameters

A MIMD, distributed memory architecture is assumed, with message-passing communications between the processors. Each processor contains an execution unit and a memory hierarchy that includes at least one level of *cache* memory. As defined in [13], a cache is the first level of memory closest to the processor. It generally has access times that are commensurate with the instruction cycle time of the processor, and is therefore several times faster than main memory access time. Cache is an associatively addressed memory which at any execution instance represents some subset of the address space of a processor. A cache can be described by several parameters denoted here by L, K, N. The parameter L describes the number of bytes per *line* of the cache. K is the number of lines per *set*, and finally, N is the number of sets in the cache. Each address generated by a processor has K possible places it can appear in the cache. If $K = 1$, then the cache is called *direct-mapped*, otherwise the cache is K-*associative*. In addition, we assume that we know the address bits used to map the set number to the K possible lines, and the address bits that select the byte within a line. We also require the knowledge of the replacement algorithm used to determine which line in the cache is to be replaced when a *cache miss* occurs.

Target Machine	Line Size	Number of Sets	Assoc.	Replace Alg.
SP1 RS/6000	64	128	4 way	LRU
SuperSPARC	32	128	4 way	Semi. LRU
i860	32	128	2 way	Random

Table 1: Target Machine Cache Parameters Used in Simulations

2.2 Definition of the computational model

We focus on scientific numerical problems in which most of the program's time is spent executing loop nests. By loop nest, we understand a perfectly nested loop structure as defined in [11]. We assume a Single Program Multiple Data (SPMD) program in which the same loop nest is executed on every processor, with each processor evaluating a subrange of the loops in the nest. Data are partitioned according to the *evaluator-owns* rule, i.e., the processor stores locally data that it computes.

2.3 Data partitioning vs. cache performance

One method to improve performance of stencil computations is to select an optimum *partitioning* of loop ranges across the set of processors. This selection requires the estimation of the execution time for the subrange assigned to each processor and communication time for necessary messages. For example in [1], it can be seen that the most efficient partitioning scheme is dependent on the domain size. Our sample scientific numerical problem is the simple but commonly used Jacobi method for solving partial differential equations. The general form of the function involved is:

$$u_{i,j,t+1} = \mathcal{F}\left(u_{i,j+1,t} + u_{i+1,j,t} + u_{i,j-1,t} + u_{i-1,j,t}\right)$$

Figure 3 shows an example of how the Jacobi code can be coded in the Fortran-D programming language for SPMD execution. The code specifies that each of the arrays A and B is sliced into four square blocks and each block is assigned to a processor. Each processor is therefore responsible for computing an array of 2048×2048. The data dependencies in the computation require that each processor exchange with other processors the edge rows or columns of the partitioned array that they share. Thus, in the compiled

```
REAL A(4096, 4096), B(4096, 4096)
PARAMETER (n$procs = 4)
DECOMPOSITION D(4096,4096)
ALIGN A, B with D
DISTRIBUTE D(BLOCK:BLOCK)
do k = 1, 100
   do j = 2, 4095
      do i = 2, 4095
         A(i,j) = (B(i-1,j) + B(i+1,j) +
                   B(i,j-1) + B(i,j+1))/4
      enddo
   enddo
   do j = 1, 4096
      do i = 1, 4096
         B(i,j) = A(i,j)
      enddo
   enddo
enddo
```

Figure 3: Parallel Fortran-D Program for Jacobi Method

code, there is a communication phase, where the edge elements are exchanged, and a computation phase, where the new values of the elements are computed. The computation phase is primarily represented by the loop nest $[j, i]$ and it is this structure we examine for cache performance in this paper. It is important to observe that data partitioning is decided by problem size, the number of available processors, and the communication overhead. Therefore the subrange allocated to each processor is independent of cache performance.

We measure the performance of this code on each processor in the number of array elements (data points) evaulated per second as a function of the total number of elements. Figure 4 shows the parallel execution performance of this program on the IBM SP1 for three different strategies of data partitioning: by row, by block, and by column.

There are two observations that are important to make about Figure 4. The first is, not unexpectedly, that the machine's performance on the problem is determined by the size of the problem, and the data partitioning used. The second one is that which partitioning is the best is dependent the data sizes. However, the sharp drop at the range value of 1024^2 for the block partition cannot be explained by a linear model of program performance or by the small increase in message size. This sharply non-linear performance curve can be explained by the cache performance of the SP1, and corresponds exactly to the single-processor performance shown in Figure 1. For the Jacobi program, on

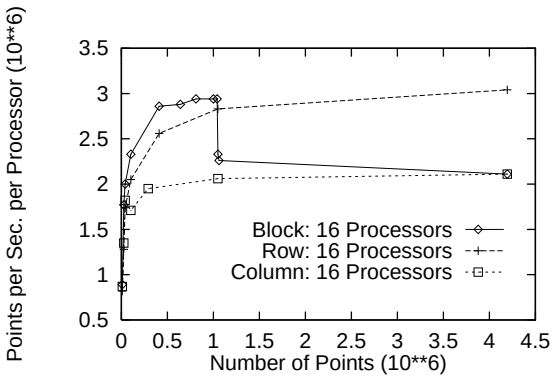

Figure 4: SP1 Multi-Processor Jacobi Performance using Standard Decompositions

the SP1, it turns out that the critical dimension is the column length, represented by the i loop. When this loop's range is larger than 1024 the performance drops off dramatically. This accounts for the lower performance of the *column* decomposition, and the higher performance of the *row* decomposition. This type of performance degradation based on loop range sizes is evident on other common numerical codes on this and other processor architectures.

Memory access patterns are induced by the order of array traversals inside loop nests. One avenue of optimization is to reorder the array traversals to obtain a more efficient memory access pattern with respect to cache performance for a given architecture. This reordering can be performed, for example, by *loop interchange* or *iteration-space blocking*. A prediction method that determines the cache miss rates for each optimization could guide the selection of the most efficient ordering, or proper iteration blocking.

Recall that in Figure 4 a decrease in performance caused by cache effects near the problem size 1024×1024 data points per processor is present. Proper code tuning, using iteration blocking as shown in Figure 1, removes this effect.

An iteration space blocking optimization exhibits improved performance because it improves locality of reference for problem sizes exceeding the loop range threshold values.

3 Cache performance estimation via simulation

This section details the cache simulation system. Section 3.1 describes operation of the cache simulator and its use of the cache parameters. Section 3.2 explains the translation of a source loop nest to a state machine. It is this state-machine that generates the *virtual addresses* which are the input to the cache simulation engine.

3.1 Cache simulator engine

The simulation engine is a parameter driven cache simulation model. It assumes that the bits that select a byte within a cache line are the lower $\log(L)$ bits of an address, and that the next $\log(N)$ bits above the line bits select the set.

The cache is initialized with a state in which all lines are *empty*. Therefore, all initial references will cause a cache miss. This is not an unreasonable assumption for two reasons. First, the likelihood of reuse of the cache from one nest loop nest to another may be small, and second, the simulation is run sufficiently long so that the number of references to each cache line is large enough for the initial misses to be insignificant.

In one cache simulation cycle, the simulation engine calls a state-machine, which was constructed from the source code to be simulated, to obtain the *next* virtual address. This address is then decoded into the *set* and *tag*. Based on the associativity of the target machine, a simulated cache tag directory is searched for the tag. If there is a match, then a hit is recorded. To model the replacement algorithm, extra state information is recorded for each set. Commonly the algorithm is a form of Least Recently Used (LRU) replacement. Thus, for a hit, the set state information is updated such that the line number within the set is put at the end of the replacement list. If the reference is a miss, a replacement line is selected, based on the set the address maps to and the LRU state (or other relevant to the particular replacement algorithm) information, and the address tag is updated.

3.2 State-machine translation

The cache simulation engine requires as an input a sequence of addresses that represent the address trace of the loop nest to be simulated. As indicated before, during each simulation cycle, the simulation engine calls a state-machine to obtain the next address.

The state machine translation process takes the original loop nest and generates three components. The first component consists of variable declarations that represent the loop variables. There is one simulated loop variable for each actual loop variable. A variable called *ping* is also allocated. This variable contains the *state* of the state-machine. The final set of variables allocated, one per array in the loop nest,

represent the virtual base address of each array respectively.

The second component generated by the translation process is an initialization routine that is called once at the beginning of the simulation. The routine initializes the ping variable, the simulated loop variables, and the virtual base addresses of the simulated arrays.

The last component generated by the translation process is the actual simulation state-machine. There are two essential parts, the first part contains the framework for the state-machine, primarily a *switch* statement that directs the operation of the state-machine based on the ping variable. The second part consists of *case* statements that are generated for each simulated loop control variable and for each unique array reference. The body of the loop control variable case statements simulate the operation of a loop statement. For array reference case statement bodies, the *rhs* virtual addresses are generated first, followed by the *lhs* virtual addresses.

```
int ping, addra, addrb, i, j, Colsz;
jacobi_init()
{
        ping = i = j = 0;
        addra = START_A; addrb = START_B;
}
jacobi_next_addr()
{
  int val;
label:
  switch(ping) {
  case 0: i = i + 1;
    if(i == Colsz) i = 0;
      ping = 1;
  case 1: j = j + 1;
    if(j == Colsz) {
      j = 0; ping = 99;
      break;
    }
    ping = 2;
  case 2: val = addrb + (8*((i-1)*Colsz) + 8*(j)); break;
  case 3: val = addrb + (8*(i*Colsz) + 8*(j-1)); break;
  case 4: val = addrb + (8*((i+1)*Colsz) + 8*(j)); break;
  case 5: val = addrb + (8*(i*Colsz) + 8*(j+1)); break;
  case 6: val = addra + (8*(i*Colsz) + 8*(j)); break;
  }
  if(ping == 99) {
    ping = 0; goto label;
  }
  ping++;
  return val;
}
```

Figure 5: Jacobi Benchmark State Machine

Each virtual address is generated by an expression that uses the simulated loop control variables and the size of the data elements of the array involved to compute, using either row-major or column-major form, the simulated offset of the data item from the beginning of each array respectively. This offset is added to its corresponding base virtual address. However, only unique addresses at each nesting level are included. This represents the likely compiler optimization of allocating a register for an array access and replacing additional references at the same nesting level with register operations.

The ping state variable is incremented to indicate that the next time the state machine procedure is called the appropriate next case statement will be active, and finally the virtual address is returned. Figure 5 shows the effect of the translation process on the first nested loop shown in Figure 3.

4 Method validation

To validate our cache simulation method we have experimented with different benchmark algorithms as well as several different architectures. The following benchmarks were used: *Vector Product* that computes a double precision vector inner product, *Jacobi Iteration* for solving partial differential equations (PDEs), *Ocean Model* solving a continuity equation taken from two-layer, linear unidirectional model of wind-driven circulation in a density-stratified ocean, and *Shallow Water Model* which is a section of the weather prediction program written at the National Center for Atmospheric Research. All benchmarks, except the first one, are stencil computations that are representative of a large class of supercomputer applications.

4.1 Experiments predicting performance

Figure 6 and the simulation results in Figures 10 and 8 show densely sampled curves of cache simulation. The plots show normalized *hit-rate* curves for each benchmark with the SP1 RS/6000 as the target architecture. The hit-rate is normalized against the best hit-rate over the range. Problem size is defined as the number of array elements allocated per processor along a single dimension, and the plots are sampled at intervals of approximately 50. For one-dimensional problems, such as the ocean model and vector product benchmarks, these are the raw number of data elements. For two-dimensional problems such, as the Jacobi iteration and shallow water model benchmarks, this is the square root of the number of elements allocated per processor.

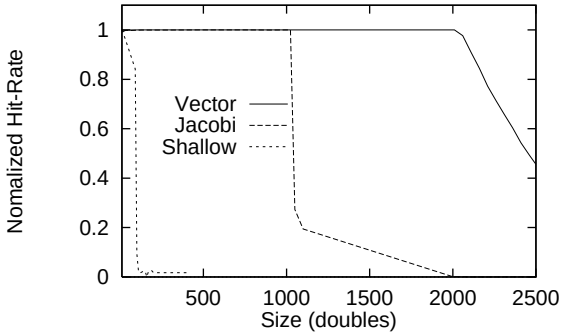

Figure 6: Cache Simulation Results for the SP1

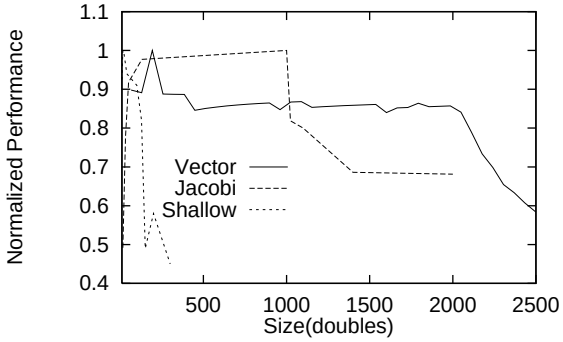

Figure 7: Benchmark Results for the SP1

Single processor performance is defined as a function of the number of data points processed per second of wall clock time. Densely sampled curves of single processor performance, normalized against the best performance obtained for each benchmark, are shown versus problem size in Figure 7 and by the target curves in Figures 10 and 8.

Comparing the figures, there is a strong correlation between the performance on the target machine and the simulated cache hit-rate. The densely sampled cache simulation results are compared with the densely sampled target performance data in Table 2. The *Actual* column in the table contains observed performance loop range thresholds, which, when exceeded, result in substantial performance degradation. The *Error(percent)* column is the relative error between the simulations and measured performance loop range thresholds on the target architecture. The maximum (in magnitude) relative error observed is 17% for the shallow water model benchmark on the SuperSPARC. Such large error is partially due to the relatively continuous decline of the performance of this benchmark and its quite low simulated miss rates. For other benchmarks, the relative error does not exceed 10%.

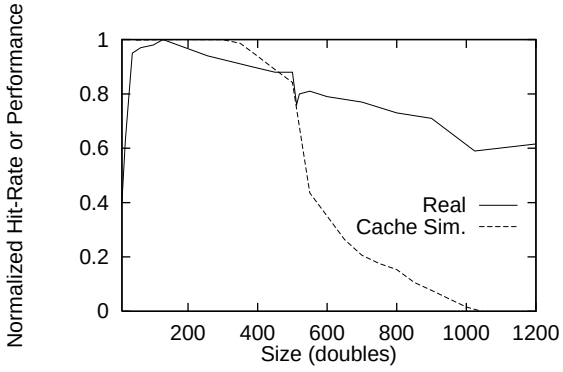

Figure 8: Simulation & Benchmark Results: i860

Target	Benchmark	Actual	Cache Sim.	Error %
SP1 RS/6000	Vector	2048	2048	0
SP1 RS/6000	Jacobi	1024	1024	0
SP1 RS/6000	Ocean	500	500	0
SP1 RS/6000	Shallow	100	90	−10
SuperSPARC	Vector	1024	1024	0
SuperSPARC	Jacobi	450	500	10
SuperSPARC	Ocean	280	260	−8
SuperSPARC	Shallow	50	60	17
i860	Jacobi	500	512	3

Table 2: Actual and Simulated Performance Loop Range Thresholds

4.2 Experiments predicting optimization results

The next set of experiments focuses on predicting the performance of original and optimized loop nests in order to evaluate usefulness of the optimization. Examples of both the *loop interchange* and *iteration-space blocking* optimizations are presented.

Loop interchange. The first example focuses on a loop interchange optimization. Figure 9 shows the performance results of the Jacobi iteration benchmark running on a single node of the SP1 and the cache simulated hit-rate. The cache simulation correctly shows that the original loop nest is preferred.

The loop interchange results for the ocean model benchmark are shown in Figure 10. Both loop nests have roughly the same performance curves and for both a decrease in performance begins approximately at a loop range value of 500. Correspondingly, the simulated cache hit-rates follow roughly the same pattern,

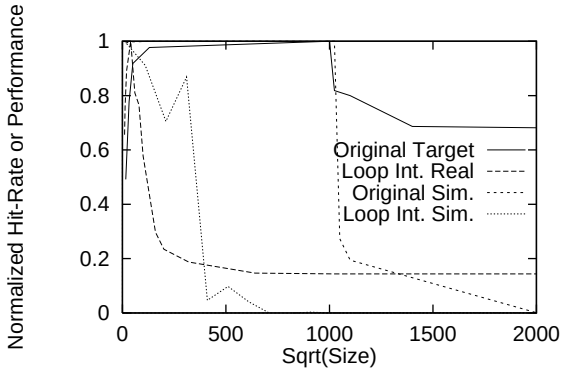

Figure 9: Jacobi Loop Interchange: Simulation and Target Results for the SP1

where both the original and interchanged loops have a distinct fall-off at the loop range of approximately 500.

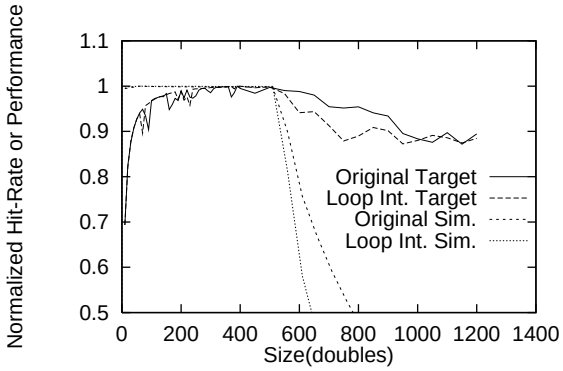

Figure 10: Ocean Loop Interchange: Simulation and Target Results for the SP1

Iteration-space blocking. One of the most important optimizations for loop nests is blocking or *tiling* [15]. This kind of optimization is used to increase the locality of data references during the loop execution to increase probability that these references will be already located in cache. To achieve this effect, additional levels of loops are added so that inner loops iterate over blocks of the original iteration space. One of the issues in blocking is to choose the blocking factor, that is the size of the blocks that maximize cache effectiveness. Choosing too small a blocking factor causes the loop to incur large numbers of *intrinsic misses* (the addresses currently in cache are used again in the next blocks but are removed from cache due to the references to the subsequent rows of the current block). Too large a factor causes a large increase in *self-interference misses* in the cache (the addresses currently in cache are used again in the following rows of the current block but are removed from cache by references to the subsequent elements in the current row) [9].

We can use the cache simulation to determine the effective blocking factor. In the following example, we take the original loop ordering of the shallow water model benchmark used in the previous sections and show that by using the optimum loop range found by inspecting the simulated cache performance profile we can improve the performance of this benchmark.

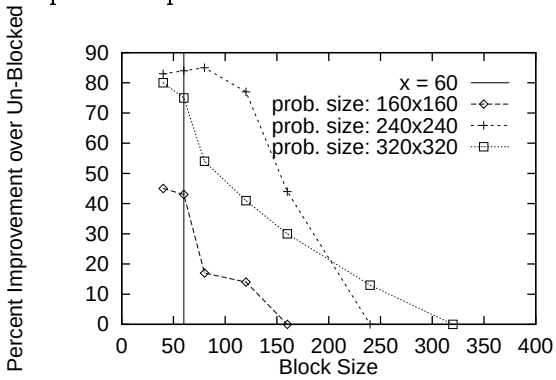

Figure 11: Performance Improvements for Various Block Sizes

From Table 2, we can predict that the optimum range value is 90 for SP1 architecture. However, this value was determined from a cache simulation that assumed that the arrays referenced inside the loops were contiguously allocated. When the code is blocked, the reference pattern generated by the blocked traversal will not follow a traversal of a contiguously allocated array. The static prediction methods, such as proposed by Wolf and Lam[9] or Fahringer[3] do not take this into consideration, and therefore they are not able to choose the correct blocking factor for a loop (in particular, the method in [9] relies on use of *copy* optimization in which each block is copied to an auxiliary array). Our approach is to determine the optimum blocking factor for *in-place* execution. The optimum in-place blocking factor is determined by modifying the virtual address generating state-machine to produce addresses representing a traversal of a block within a larger array. The state-machine is modified by changing the array address calculation so that a distance between rows (in a row-major organization) is larger than the row size of the block. When this is done, the cache miss-rate increases and the optimum loop range decreases to 60 for the shallow water benchmark run on SP1.

Figure 11 shows the target execution performance improvement for blocked versions of the shallow water model benchmark over the original code. Three

different problem sizes are shown, each executed with several different block sizes. The predicted blocking size of 60 yields performance within a few percent of the optimum performance improvement for each of the problem sizes.

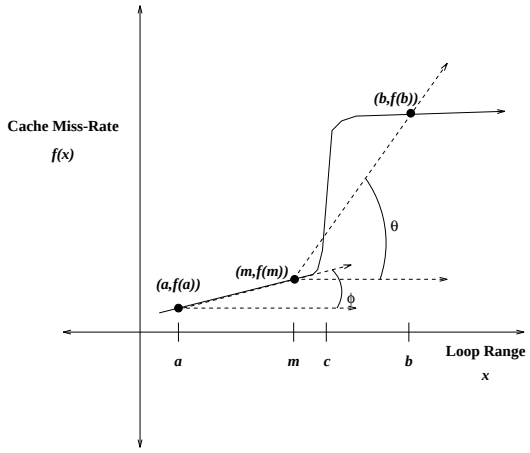

Figure 12: Values Computed during the Heuristic

5 Heuristic search method

Unfortunately, developing a densely populated hit-rate curve via simulation is time consuming, hence it is unsuitable for integration into a compilation system. However, there is both theoretical and experimental evidence that the cache miss-rate, (*miss-rate*= 1−*hit-rate*), curve as a function of a relevant range is *S shaped* as seen in Figure 12.

For performance reasons it is most desirable to operate at the rightmost value of the loop range, x, to the left of the steep region approaching c. This is the maximum problem size exhibiting good performance characteristics (in the case of loop transformations, smaller values would increase the iteration overhead with respect to the number of computations performed). The abrupt change takes place over very small range intervals, on the order of 10^1.

Taking advantage of the shape of the cache-miss rate curves, we developed the heuristic, for finding the optimal range (see Figure 13). The user can define also a tolerance τ on x, and limit γ for the left and right slope values (default values are $\tau = 10$, $\gamma = 0.1$). The function, $f(m)$, returns a normalized simulated cache miss-rate value with respect to the initial range. The required number of steps for the recursive bisection is bounded by $\log \frac{b-a}{\tau}$. Hence, the selection of a good initial interval is important. We plan to use static cache performance analysis techniques for selecting initial values for a and b.

```
int h(int a, int b)
{
    int m;
    m = (a + b)/2;
    φ = (f(m) - f(a))/(m-a); θ = (f(b) - f(m))/(b-m);

    if(b < a + 2*τ || θ > γ)
        return m;
    else
        if (φ > γ) return a;
    if (θ < φ) return h(a,m) else return h(m,b);
}
```

Figure 13: The Heuristic Algorithm

A comparison of actual range values with the heuristic results is presented in Table 3. The *Actual* column contains observed performance loop range thresholds which, when exceeded, greatly decrease target performance. The *Heuristic* column are values reported by the heuristic. The *Error(percent)* records the relative error (in percent). The *Time(sec.)* column shows the running time of the heuristic on a uniprocessor SUN Sparc-10 computer. The maximum time to evaluate the heuristic search is 50 seconds for the shallow water model benchmark with the SuperSPARC as a target. The typical running time of this kind of programs is measured in hours, if not days, so even a few percent improvement in the running time can exceed the above compile time penalty many times over. A positive value of an error in Table 3 indicates that the heuristic picks a larger performance loop range threshold than the actual threshold. The positive error is usually small, not exceeding 8%. The largest negative error is the Jacobi iteration benchmark on the i860 with a 20% error. This occurs because of the way the heuristic selects the threshold, since it correctly discovers that the miss rate increases with a sharp jump occurring around problem size of 512. Intentionally, the heuristic underestimates the threshold value by picking the left edge of the upward slope, becasue the cost of underestimating is a slight increase in loop control overhead, and overestimating can increase the miss-rate and degrade performance considerably.

6 Conclusions and further research

We have shown that it is possible to efficiently determine the performance characteristic of loop nests over a loop range by using an heuristically-directed

Target	Bench-mark	Actual	Heur-istic	Error %	Time sec.
RS/6000	Vector	2048	2056	4	30
RS/6000	Jacobi	1024	967	−6	17
RS/6000	Ocean	500	503	1	10
RS/6000	Shallow	100	86	−14	50
S. SPARC	Vector	1024	1032	1	20
S. SPARC	Jacobi	450	487	8	5
S. SPARC	Ocean	280	252	−10	9
S. SPARC	Shallow	50	56	−12	50
i860	Jacobi	500	400	−20	8

Table 3: Heuristic Cache Search Performance

architecturally-based cache simulation. We have also shown that this information can successfully be used to guide common loop optimization decisions, such as loop-interchange and iteration-space blocking. Finally, we have found that to determine the correct blocking factor, the virtual addresses generated for the simulation must represent access in a non-contiguously allocated array.

We need to more carefully choose the number of virtual addresses generated for each simulation. So far, we have found that the simulated miss-rate characteristic of stencil-based numerical codes to be consistent over a wide range of numbers of virtual address generated once a threshold trace length has been reached. The goal would be to select an heuristic that provides a long enough trace for accurate simulation, but not too long as to be overly time-consuming.

One possible improvement to the simulation state machine itself is the addition of other likely compiler optimizations such as common sub-expression elimination, loop invariant or inductive expressions when determining the state-machine simulated references to more accurately represent the reference pattern of actual compiled code.

Our next effort is to integrate static performance estimates of these optimized loop constructs based on a *training-set*-like method [1], adjusted for cache effects. Once this is done we will have an accurate execution time function that describes the loop. Additionally we will integrate it with a method to determine the cost of the communication requirement of the loop. This will result in a fully characterized program components that can be used for both compiler optimization decisions and machine architecture evaluations.

This research is part of an ongoing effort to provide an architecturally based parallel program optimization tools built around the EPL [10] system.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide data Partitioning Descisions. In *Third ACM SIGPLAN Symposium on Principles and Practice on Parallel Programming, Williamsburg, VA*, pages 213–223. ACM, NY, April 1991.

[2] M. J. Clement and M. J. Quinn. Analytical Performance Prediction on Multicomputers. In *Proc. ACM Supercomputing, Seattle, WA*, pages 886–894. ACM, NY, August 1993.

[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. Gerasoulis and T. Yang. Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *Journal of Parallel and Distributed Computing*, (16):276–291, 1992.

[6] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993.

[7] J. Ferrante, V. Sarkar, W. Thrash. On Estimating and Enhancing Cache Effectiveness. In *Languages and Compilers for Parallel Computing, Fourth Internation Workshop, Santa Clara, CA*. Springer-Verlag, NY, August 1991.

[8] K. Kennedy and K. S. McKinley. Optimizing for Parallelism and Data Locality. In *International Conference on Supercomputing 1992, Washington, D.C.*, July 1992.

[9] 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.

[10] C. Özturan, B. Sinharoy, and B.K. Szymanski. Compiler Technology for Parallel Scientific Computation. *to appear in Scientific Programming*, 1994.

[11] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, Houston, Texas, May 1989.

[12] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research monographs in parallel and distributed computing. MIT Press, Cambridge, MA., 1989.

[13] H. S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 1990.

[14] O. Teman, C. Fricker, and W. Jalby. Impact of Cache Interferences on Usual Numerical Dense Loop Nests. *Proceedings of the IEEE*, 81(8), August 1993.

[15] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proc. ACM SIGPLAN, Toronto, Canada.* ACM, NY, June 1991.