Parallel Processing Letters

Parallel Processing Letters, vol. 6, no. 1, March 1996, pp. 173-184

Program Optimization Based on Compile-Time Cache Performance Prediction

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

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

1. Introduction

Loop nests are prime prospects for code optimization. Many loop restructuring techniques influence cache performance. Examples are loop interchange, fusion, distribution, iteration-space blocking, and skewing (c.f., [4,5,6]). When applying such techniques it is important to select the loop range that maximizes the cache performance. Choosing too small a range increases *intrinsic misses*, whereas too large a range increases *self-interference misses*.

In this paper we describe a compile-time method for determining cache performance on a generalized loop range. We also present a heuristic for finding optimal loop ranges for such nests. Finally, we demonstrate results of applying this technique to several application codes and processors.

1.1. Previous Work on Cache Performance Prediction

Two components are needed to predict the performance of the cache: (i) the reference string of an application, and (ii) actions of the cache over the string. Depending on how these components are created or approximated current methods fall into the following three classes: (1) exact, (2) execution-driven simulation, and (3) analytical models. The method presented in this paper belongs to a new class, named *compile-time simulation*. The relationships among these classes are shown in Figure 1.

Exact Methods generate the reference string by creating a synthetic mix or abbreviated form of the application code, executing it, and running it through the real cache (cf. [7]). Although actions of the cache over the string are very fast (they

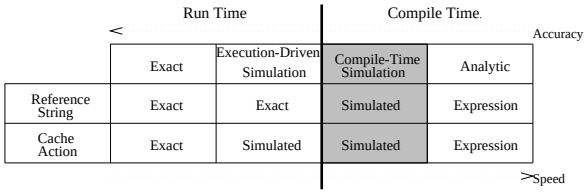

Fig. 1. Relationship of Cache Performance Prediction Classes

are taken by the actual hardware), the method requires repetitive compilations and executions of the application codes; therefore it is too time-consuming to be used at compile-time.

Execution-Driven Simulation described in [8,9,10], runs a model of a cache over the reference string generated by an execution of an application. Such a solution allows for mapping from a reference to the source code. Thus, the programmer can be informed of the number and types of cache misses caused by each source line. However, the cache model is significantly slower than the actual cache. The time for repeated compilation, instrumentation, and execution of the code needed while searching for the optimal loop range, renders this method impractical at compile-time.

Analytical models approximate both the reference string and the actions of the cache. Instead of looking at individual references, these methods determine the number of distinct cache lines referenced in a loop nest based on the number of reference classes [4,5] or array classes [11]. This number combined with the number of cache lines and the the values of the loop ranges yields a crude approximation of the cache hit-rate. The method cannot accurately account for effects of different cache line replacement algorithms, context switches, set-associative cache mappings, or translation of virtual to physical addresses. It is also difficult to accurately attribute cache behavior to the source code of the loop nest. However, the analysis relies only on the syntax of the loop nests and formulas involving parameters of a cache. Thanks to that, analytic methods are very fast and suitable for use at compile-time.

Compile-Time Simulation is introduced in this paper as a new class. Figure 1 shows the relationship of this class to others. Compile-time simulation differs from execution-driven simulation in the way the reference strings are generated. In our method, the definition of the reference string is created during parsing of the source code, as opposed to generating the reference string by compiled code execution. Both execution-driven and compile-time simulation can accurately account for different cache-line replacement algorithms and different cache organizations (direct, set-associative, *etc.*). Context-switch effects can be approximated by the periodic clearing of the current contents of the cache. Instead of physical addresses, both models use compile-time addresses that change during loading. However, rarely a cache miss is caused by loading [10], so the effect of this approximation on the hit-rates is negligible. The results of the compile-time simulation are less accurate than execution-driven simulation because the simulated reference string is just an

approximation of the real one (*e.g.*, assignment statements guarded by conditionals are assumed to always execute). In our experiments we show that these results are accurate enough for cache performance prediction.

The advantage of the proposed method is that the compact representation of the reference string is created at compile-time immediately after parsing. This representation can be modified during optimization to simulate different loop ranges without the need for source program recompilation or execution.

In the following section we show how the reference string is defined from the source program and how it is used to obtain the loop ranges for which the cache performance is optimized. Section 3 describes the use of this method for loop optimization for different architectures, and provides the timing results of its application. Section 4 suggests future work.

2. Compile-Time Simulation Method

We describe here (i) a technique for generating the reference string from the source code at compile time, and (ii) a simulation model of a cache described in terms of basic cache parameters. In addition, we present a heuristic search algorithm for finding the loop ranges that optimize cache performance.

2.1. Reference String Generation

As shown below, the parser of a source language can be extended to produce an expression that can be used to quickly generate the reference string. Following our interest in parallel program optimization, we focus on programs expressed as nests of loops. To keep the source code translation language-independent, we will discuss the parser action for a language, $\mathcal{L}$, that defines a sequence of loop nests:

$$\mathcal{L} = \{p \leftarrow c, c \leftarrow l\ c \mid A\ c \mid l \mid A, l \leftarrow HEAD\ c\ TAIL\}$$

where non-terminals *p,c,* and l stand for a program, a sequence of statements, and a loop, respectively. Terminals *A, HEAD,* and *TAIL* represent an assignment statement, the head of a loop statement, and the closing of a loop statement, respectively. The syntax of the terminals is language dependent.

By augmenting a parser with proper actions, it can easily generate the following functions over the symbols in $\mathcal{L}$:

$$C(s) = \begin{cases} \text{number of references} & \text{if } s = A \\ \text{number of statements} & \text{otherwise} \end{cases}$$

$$S(s,i) = \begin{cases} \text{the } i^{th} \text{ reference of } s & \text{if } s = A \\ \text{the } i^{th} \text{ statement of } s & \text{otherwise} \end{cases}$$

$$range(s) = \begin{cases} \text{the range of loop} & \text{if } s = l \\ 1 & \text{otherwise} \end{cases}$$

The range of a loop is defined at compile-time if it is either a constant or an expression over the control variables of outer loops with defined ranges. Because the described cache prediction method is intended for loop range optimization (see Section 2.3), we are interested only in an upper bound for the optimal loop range for the given cache. Such an upper bound can be estimated from the loop body

and the cache line size (see below). Thus, without loss of generality, we can assume that the range of a loop can always be established at compile-time.

We define the *reference space* of a symbol $s \in \mathcal{L}$ as:

$$\mathcal{R}(s) = \begin{cases} \bigcup_{j=1}^{C(s)} \{j\} & \text{if } s = A \\ \bigcup_{i=1}^{range(s)} \{i\} \times \bigcup_{j=1}^{C(s)} \{j\} \times \mathcal{R}(S(s,j)) & \text{otherwise} \end{cases}$$

$\mathcal{R}(s)$ is a set of vectors of varying length. Elements of these vectors assume integer values from 1 to the number of statements in a corresponding sequence of statements, or to the range of a corresponding loop. Hence, we will use a sub-vector of a vector v, denoted v_s, that consists of those elements of v that select a statement from a sequence of statements. For each vector $v \in \mathcal{R}(p)$ we define: $len(v) = |v|$, and for a pair of vectors, $v_1 \neq v_2 \in \mathcal{R}(p)$:

$$d(v_1, v_2) = \min_{j \leq len(v_1)} (v_1[j] \neq v_2[j]) \quad v_1 \prec v_2 \Leftrightarrow v_1[d(v_1,v_2)] < v_2[d(v_1,v_2)]$$

The function d is well defined because it follows from the definition of $\mathcal{R}(p)$ that for any pair of vectors $v_1, v_2 \in \mathcal{R}(p)$ if $\forall i \leq len(v_1) : v_1[i] = v_2[i]$ then $len(v_1) = len(v_2)$. The relation $\prec$ defines the lexicographic (total) order in $\mathcal{R}(p)$. The usual definition of the successor of an element $v \in \mathcal{R}(p)$ applies:

$$v_+ = succ(v) \quad \Leftrightarrow \quad (\forall v' \in \mathcal{R}(p) : v \prec v' \Rightarrow (v_+ = v' \mid v_+ \prec v'))$$

Figure 2a shows an example of a sequence of $\mathcal{R}(p)$ elements in lexicographical order for the program in Figure 2b. The function $G(v_s, i)$ defined for any $v \in \mathcal{R}(p)$ returns the statement pointed out by the first i elements of v_s:

$$G(v_s, i) = \begin{cases} S(p, v_s[1]) & \text{if } i = 1 \\ S(G(v_s, i-1), v_s[i]) & \text{otherwise} \end{cases}$$

Figure 2b shows v_s projected from v, and the relation of elements of v_s to statements at different levels. The last element of v_s always refers to a variable reference.

For each program variable a, let the function $n(a)$ define its dimensionality ($n(a) = 0$ if a is a scalar). Any reference to a is then indexed by $n(a)$ arithmetic expressions. In many scientific programs the indexing expressions are functions over control variables of loops surrounding the reference. For a reference $G(v_s, len(v_s))$ defined by the vector v, let $var(v)$ denote the referenced variable and $f(v,i)$ denote the i indexing function, where $0 < i \leq n(var(v))$. With these definitions, the virtual address, *vaddr*, generated by the reference defined by vector v, is given by:

$$vaddr = addr(var(v)[f(v,1), \ldots, f(v,i), \ldots, f(v, n(var(v)))]),$$

where *addr* translates the vector of $n(var(v))$ index values for a variable $var(v)$ into a virtual address in the linear memory.

The presented translation can be done if the indexing function involves only loop control variables as arguments and constants as coefficients. More general and still translatable indexing expressions can involve loop constants (*i.e.*, variables

which are not assigned to in the loop body). In such a case, however, the exact virtual address will not be known, so the simulation will be exact only for fully associative caches. For the case of indexing expressions with indirect mappings or with variables that are reassigned in the loop body, we do not produce any virtual addresses for the corresponding references. The justification of this treatment is that such references change from one loop iteration to the next so their effect on cache performance is rarely dependent on the loop range. Ignoring these kind of references usually will not have a significant impact on the determination of the optimal loop ranges.

It is clear from the above definitions that the reference string is produced when the virtual address translation is applied to the subsequent successors of the minimum vector in $\mathcal{R}(p)$. Because an efficient implementation of the successor function is paramount to the speed of the cache simulation, we consider an alternative definition of this function through the values of the corresponding $C(s)$ and $range(s)$ functions. Let $R(v,i)$ stand for $range(G(v_s,i))$ if $G(v_s,i)$ is a loop and for $C(G(v_s,i))$, otherwise, where $i \leq len(v)$. Then, the successor can be defined as follows:

$$succ(v)[i] = \begin{cases} v[i] & \text{if } i < last(v) \\ v[i]+1 & \text{if } i = last(v) \\ 1 & \text{otherwise,} \end{cases}$$

where $last(v) = max_{i \leq len(v)}\{v[i] \neq R(v,i)\}$ is the last position in vector v that is not ready to overflow. The last element of the vector $succ(v)$ is such that $G(succ(v)_s, len(succ(v)_s))$ points to a variable reference. This definition allows for a fast generation of the reference string for the given program p.

Figure 2 shows an example of the generation of a set of references from a program in Figure 2b. Consider the relationship of the next-to-last reference in Figure 2a (rewritten as v in Figure 2b) to the source. The first element of the reference is 1 which indicates the entire program, p. The solid lines show the statements selected by each component of v_s. The dotted lines represent the iteration counters for the selected loops. Figure 2c shows the action of the $succ(v)$ function. In this case, $last(v)$ points to the third element of v. This is because $2 < range(I_1)$, and for all i greater than 3 we are at the last variable of the last statement of the last iteration of the I_3 loop nest. The successor of v therefore points to first variable of the first statement of the I_1 loop, with I_1 incremented by one. The representation of the entire reference string is very compact. For the example in Figure 2 only 10 values of $R(v,i)$ are required. This contrasts with the entire reference string which, assuming $I_1 = I_2 = I_3 = 1024$, contains about $7M$ references.

2.2. Cache Model

As defined in [7], a cache is the first level of memory, fast and closest to a processor. At any execution instance it represents some subset of the address space of a processor. Our cache model is characterized by a *replacement* algorithm and the following three parameters: L is the number of bytes per *line* of the cache, K is the number of lines per *set*, and N is the number of sets in the cache. The

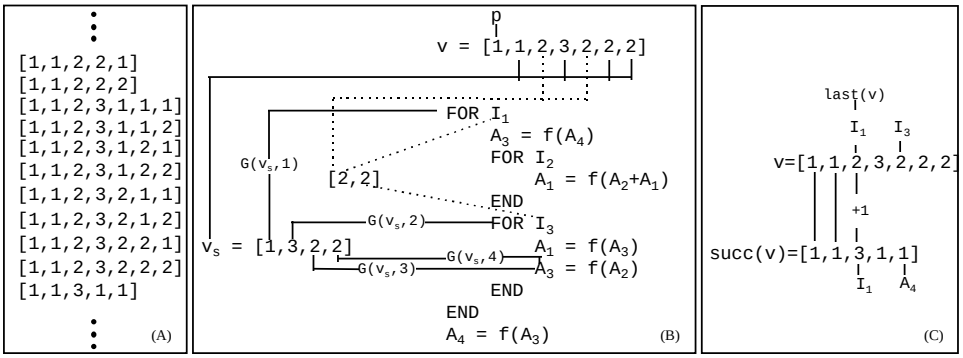

Fig. 2 Example Vector Space and Program

implementation of the cache model assumes that the lower $\log(L)$ bits of an address select a byte within a cache line, and the next $\log(N)$ bits select the set. The cache is initialized with all lines marked *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 loop nest to another is usually very small. 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 obtains the lexicographically next element of the reference space of the loop nest to obtain the *next* virtual address. This address is then run through the cache algorithm, and the action of the cache is recorded.

2.3. Heuristic Search

Generating a densely populated cache *hit-rate* curve via simulation is time consuming, and therefore unsuitable for integration into a compilation system. However, theoretical and experimental evidence show that the cache hit-rate curve as a function of a relevant range is *Z shaped* (see Figure 3, right).

The overall program performance is highest at the rightmost value of the loop range, i_c, which is immediately to the left of the steep decline region. This range value corresponds to the maximum problem size exhibiting good cache performance. An abrupt change in cache performance takes place over a small interval (of size about 10, or smaller) of loop range values.

Taking advantage of the shape of the cache hit-rate curves, we developed a heuristic for finding the optimal range (see Figure 3, left) by recursive halving the initial interval to which the solution belongs. The initial endpoints of this interval, i_l, i_h, are established as follows. The lower bound on value of i_l is defined by the loop statement overhead, so it is inversely proportional to lbs, the number of operations in a single loop body execution. To keep this overhead in the range of a few percent, we set $i_l = \max(10, 500/lbs)$. The upper bound, i_h, is estimated by an analytical

method similar to the one presented in [10] using the formula:

$$i_h = \left(\frac{N \times K \times L}{\sum_{i=1}^{n_{acs}} size(i)} \right),$$

where n_{acs} is the number of array access classes (see [10] for their definition) in the innermost loop, $size(i)$ is the data size of references in the i^{th} reference class measured in bytes, and K, L, N are parameters of the cache described in the previous section.

Another important part of the heuristic is to determine if the midpoint $(i_l + i_h)/2$ is to the left or to the right of the solution. The user can modify the default values used in this determination by providing a tolerance τ on i, and limit γ for the left and right slope values (default values are $\tau = 10$, $\gamma = 0.1$). The function, $f(i)$, returns a normalized simulated cache hit-rate value for range i. The required number of steps for the recursive bisection is $O(\log \frac{i_h - i_l}{\tau})$, so only a few ranges of the loop must by tried before the optimum is found.

```
int h(int i_l, int i_h)
{
    int i_m;
    i_m = (i_l + i_h)/2;
    φ = (f(i_m) - f(i_l))/(i_m − i_l);
    θ = (f(i_h) - f(i_m))/(i_h − i_m);
    if(i_h < i_l + 2*τ || θ > γ)return i_m;
    else if (φ > γ) return i_l;
    else if (θ < φ) return h(i_l, i_m)
    else return h(i_m, i_h);
}
```

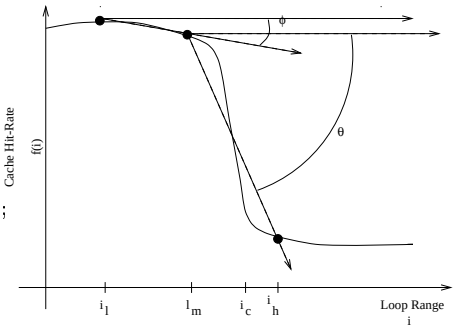

Fig. 3 Heuristic Algorithm (left) and Values computed during heuristic (right)

3. Validation of the Method

In this section we examine the accuracy and cost of the compile-time simulation method and its applicability to loop optimization for cache performance. To this end we report on a set of experiments conducted on several benchmarks described in Section 3.1.

There are two potential sources of inaccuracy to the proposed method. First, the references generated by the compile-time (see Figure 1) simulation are an approximation of the real reference string. To measure the impact of this approximation we compare densely sampled cache simulation and real performance curves for each benchmark. We show that the shapes of the these curves are similar, though not identical.

The second source of inaccuracy is the heuristic from Section 2.3 used to determine the largest range with good hit-rate performance. We apply the heuristic to the benchmarks and show that the found ranges are within several percent of the optimum. These runs also demonstrate that the times required by the heuristic driven simulations are acceptable for compile-time analysis.

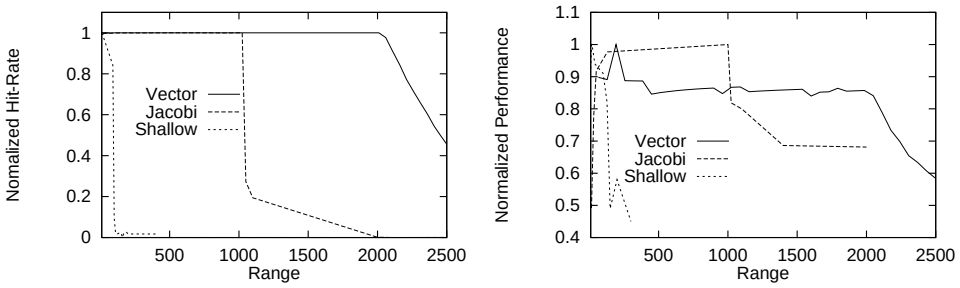

Fig. 4 Cache Simulation (left) and Benchmark (right) Results for the SP1

The final set of experiments reported below focuses on the use of range returned by the heuristic in guiding two common loop optimization techniques: loop-interchange and iteration-space blocking.

3.1. Benchmarks

We drew our benchmarks from the following applications: *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. We believe that these codes are a representative sample for a large class of supercomputer applications. Loop bodies in these codes use affine indexes and loops are nested up to three levels deep.

3.2. Compile-Time Simulation verses Actual Performance

Figures 4, 5, and the right hand graph in Figure 6 show densely sampled curves of cache simulation and benchmark performance results. It should be noted that dense sampling of simulate hit-rate curves is not part of the proposed method and was done only to verify the accuracy of the simulation. The curves represent *hit-rate* curves for each application and benchmark performance, each normalized against the best hit-rate and the best performance over the range, respectively. The range represents the size of the problem for one-dimensional problems and the size of each dimension of the problem for two-dimensional problems. The plots are sampled at intervals of approximately 50.

These experiments show that there is a strong correlation between the shape of the performance on the target machine and the shape of simulated cache hit-rate. Additionally, we used the dense curves to locate i_c, the largest range to the left of the steep decline region of the curve. The results are presented in the left part of Table 1, where the first *Error(percent)* column shows the difference between the *Actual* and *Cache Simulation* results. The maximum (in magnitude) relative error observed is 17% for the shallow water model benchmark on the SuperSPARC. Such a large error is caused in this case by the relatively smooth decline of the performance of this benchmark and its quite high hit rates. For other benchmarks,

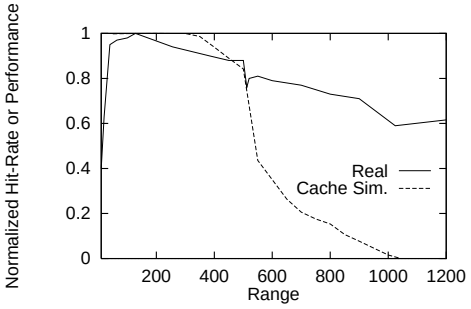

Fig. 5 Simulation & Benchmark Results: i860

the relative error does not exceed 10%.

3.3. Heuristic Accuracy and Simulation Time

To show that the proposed method is feasible for integration into a compilation system, the accuracy and time required by the heuristic are given in the right part of Table 1. The rightmost *Error(percent)* column in Table 1 records the relative error of the heuristically determined range versus the optimal value. The second *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 heuristic generally picks a range value that is within a few percent of the optimal value. Positive errors do not exceed 8%. The largest negative error, 20%, is for the Jacobi iteration benchmark on the i860. Intentionally, the heuristic underestimates the threshold value by picking the left edge of the downward slope, because the cost of underestimating is a slight increase in loop control overhead, while overestimating can move the selected range to the low hit-rate region.

Table 1. Actual, Simulated, and Heuristic Loop Range Thresholds

Target	Benchmark	Actual	Cache Sim.	Error %	Analytic	Heuristic	Time sec.	Error %
SP1	Vector	2048	2048	0	2048	2056	30	4
SP1	Jacobi	1024	1024	0	1024	967	17	-6
SP1	Ocean	500	500	0	585	503	10	1
SP1	Shallow	100	90	-10	292	86	50	-14
SP1	NC. Shallow†	65	62	4	-	60	50	2
SPARC	Vector	1024	1024	0	1024	1032	20	1
SPARC	Jacobi	450	500	10	512	487	5	8
SPARC	Ocean	280	260	-8	292	252	9	-10
SPARC	Shallow	50	60	17	146	56	50	-12
i860	Jacobi	500	512	3	512	400	8	-20

† NC indicates non-contiguous storage

Table 1 also shows the ranges determined by applying an analytic method, such as in [11]. The analytic method is generally too optimistic which results in the selection of a range in the low-performance region of the hit-rate curve. For example,

the best range determined by the analytic method for the shallow water benchmark on the SP1 will deliver about 50% of the performance for the range determined by the compile-time simulation and heuristic.

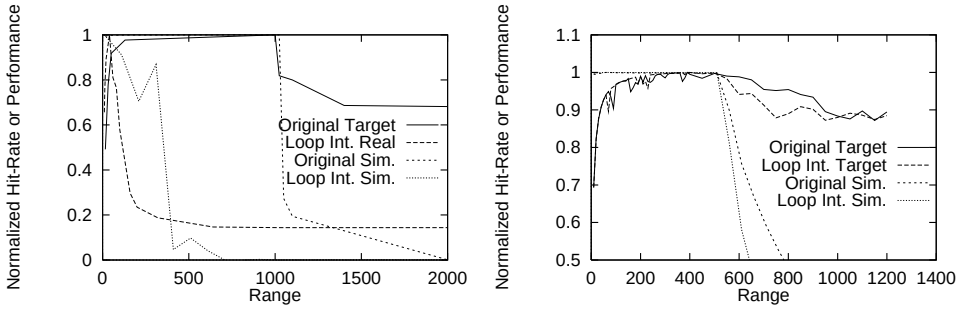

Fig. 6 Loop Interchange: Jacobi (left) and Ocean (right) Results for the SP1

3.4. Performance Prediction of Loop Optimization

Here we focus on predicting the performance of original and transformed loop nests in order to select optimal loop transformation. We show that the heuristic can be used to quickly determine the *iteration-space blocking* factor. Some results for *loop interchange* are given in Table 1 and Figure 6 (for details see [12]).

Blocking or tiling [4] is used to increase the locality of data references during the loop execution by adding additional levels of loops so that inner loops iterate over blocks of the original iteration space. Blocking can improve cache performance by increasing reuse and decreasing interference. On the right in Figure 7 we show the results of blocking the shallow benchmark using a range of block sizes for three problem sizes. By choosing the correct blocking factor the performance improves by as much as 85%.

The graph on the left in Figure 7 shows the performance improvement obtained by applying blocking to the Jacobi benchmark. Using the cache threshold value determined by the compile-time simulation improves the performance by 25% (except in a narrow region of 8% degradation).

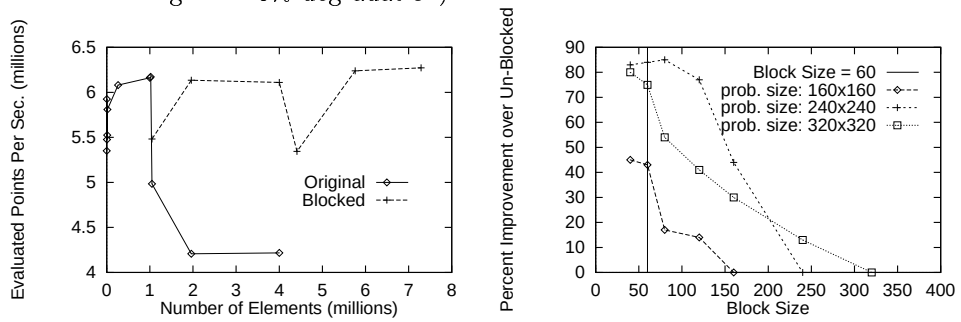

Fig. 7 Performance Improvements Due to Blocking

The reference pattern generated after blocking is different than one produced by a traversal of a contiguously allocated array. Static prediction methods, such

as proposed by [4] or [11], do not take this into consideration, and therefore they may not correctly choose the blocking factor for a loop. As shown in Table 1, using results from the analytical method in [11] a block size of 292 would be chosen. As evidenced in the right part of Figure 7, this would result in little or no performance improvement.

We can correctly determine the optimum blocking factor for *in-place* execution by modifying the simulated array addresses to represent the references generated by the traversal of a block within a larger array. This is done by offsetting the simulated block in the blocked iteration dimension. In our reference generation model this is done by adding a constant offset to $f(v, i)$ for each blocked dimension. The resulting cache hit-rate decreases (*i.e.*, i_c moves to the left), and the optimum loop range returned by the heuristic decreases from 90 to 60, (see Table 1 rows marked SP1 Shallow and SP1 NC Shallow) This block size yield improvement within a few percent of the optimum.

4. Future Work

The current simulation system takes a sequence of loop nests and represents it as a reference space with a total order. The reference string for each simulation is generated by the sequence of applications of the successor function in this space.

Instead of going through each point of a reference space we could traverse just the points that could change the contents of the cache; these are exactly the points for which cache-misses occur. Hence, instead of using a successor function, we can move to the lexicographically closest point of the reference space for which the current lines are overflowed or at which the range of one of the loop control variables is exhausted. Such an approach can speed up the performance by the factor equal to the inverse of the miss-rate. However, it requires that the indexing expressions are affine functions (otherwise, it is difficult to predict which future reference reaches beyond the current cache contents). The advantages and disadvantages of such an approach in comparison to the approach presented here are the subject of our current research.

Acknowledgements The authors would like to thank William Maniatty for help with the method evaluation, and Eva Ma, Peter Tannenbaum, and the anonymous reviewers for numerous suggestions for improving this paper. This work was supported in part by ONR grant N00014-93-1-0076, by NSF grant CCR-9216053 and by a grant from AT&T Corporation.

References

1. E. Deelman, W. Kaplow, B. Szymanski, P. Tannenbaum, and L. Ziantz, Integrating Data and Task Parallelism in Scientific Programs, in *Languages, Compilers, and Run-Time Systems for Scalable Computers*, (Kluwer Academic Publishers, Boston, MA, 1995) 169–183.
2. V. Decyk, Personal Communication.
3. A. Gerasoulis and T. Yang, On the Granularity and Clustering of Directed Acyclic Task Graphs, *IEEE Transaction on Parallel and Distributed Systems* 4(**6**)(1993) 686–701.

4. M. S. Lam, E. E. Rotherberg, and M. Wolf, The Cache Performance and Optimizations of Blocked Algorithms, in *ACM Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April, 1991, 63–74.

5. S. Carr, K. S. 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, 252–262.

6. D. Gannon, W. Jalby, and K. Gallican, Strategies for Cache and Local Memory Management by Global Program Transformation, *Journal of Parallel and Distributed Computing*, October, 1988.

7. D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, (Morgan Kaufmann, San Mateo, CA, 1993).

8. A.J. Goldberg and J. Hennessy, Performance Debugging Shared–Memory Multiprocessor Programs with Mtool, in *Proceedings Supercomputing 91*, (IEEE Computer Science Press, Los Alamitos, CA, 1991) 481–490.

9. A. Gupta, M. Martonosi, and T. Anderson, MemSpy: Analyzing Memory System Bottlenecks in Programs, *Performance Analysis Review*, 20(**1**)(1992).

10. A. R. Lebeck and D. Wood, Cache Profiling and the SPEC Benchmarks: A Case Study, *IEEE Computer*, October, 1994.

11. T. Fahringer, Automatic Cache Performance Prediction in a Parallelizing Compiler, in *Proceedings of AICA*, Lecee, Italy, September, 1993.

12. W. Kaplow, W. Maniatty, B. Szymanski, Impact of Memory Hierarchy on Program Partitioning and Scheduling, in *Proceedings of the* 28^{th} *Hawaii International Conference on System Sciences*, Maui, Hawaii, January, 1995.