A Framework for Qualitative Performance Prediction

Chung-Hsing Hsu
Ulrich Kremer
Department of Computer Science
Rutgers University

TECHNICAL REPORT

LCSR–TR363

July 1998

A Framework for Qualitative Performance Prediction

Chung-Hsing Hsu and Ulrich Kremer*

Department of Computer Science
Rutgers University

Abstract

Performance prediction models at the source code level are crucial components in advanced optimizing compilers, programming environments, and tools for performance debugging. Compilers and programming environments use performance models to guide the selection of effective code improvement strategies. Tools for performance debugging may use performance prediction models to explain the performance behavior of a program to the user. Finding the best match between a performance prediction model and a specific source–level optimization task or performance explanation task is a challenging problem. The best performance prediction model for a given task is a model that satisfies the precision requirements while including as few performance factors as possible in order to minimize the cost of the performance predictions. In optimizing compilers, the lack of such a cost–effective performance model may make the application of an optimization prohibitively expensive. In the context of a programming environment, marginal performance factors should be avoided since they will obscure reasoning about the observed performance behavior.

This paper discusses a new qualitative performance prediction framework at the program source level that automatically selects a minimal set of performance factors for a target system and performance precision requirement. In the context of this paper, a target system consists of a compiler, an operating system, and a machine architecture. The performance prediction framework identifies significant target system and application program parameters that have to be considered in order to achieve the requested precision. Such parameters may include application factors such as number and type of floating point operations, and machine characteristics such as L_1 and L_2 caches, TLB, and main memory. The reported performance factors can be used by a compiler writer to build or validate a quantitative performance model, and by a user to better understand the observed program performance. In addition, the failure of the framework to produce a model of the desired quality may be an indication that there exists a significant performance factor not considered within the performance framework. Such information is important to guiding a compiler writer or user in a more efficient search for crucial performance factors.

Preliminary experimental results for a small computation kernel and a set of twelve target systems indicate the effectiveness of our framework. The target systems for the experiment consisted of four machine architectures (SuperSPARC I-II and UltraSPARC I-II running Solaris 2.5) and three compiler optimization levels (`-none`, `-O3`, `-depend -fast`). Our prototype framework determines different performance models (1) across different precision requirements for the same target

*e-mail: chunghsu@cs.rutgers.edu, uli@cs.rutgers.edu; address: Department of Computer Science, Hill Center, Busch Campus, Rutgers University, Piscataway, NJ 08855

system, and (2) across different target systems for the same precision requirement. In addition, the prototype framework leads us to the discovery of a performance factor within the operating system by failing to produce a performance model of a requested precision.

1 Introduction

Source–level performance prediction is a key component in any source–to–source compiler or any interactive environment that allows performance debugging. Performance models are needed to guide the compiler or user in the code optimization process by supporting the decision of which optimization transformation to apply next. The ideal performance model matches exactly the decision process for an optimization or a set of optimizations, i.e., it *ranks* the optimization alternatives correctly according to their expected performance benefits. Since a more detail performance model is typically more expensive to compute, the ideal performance model is also minimal in the sense that it does not model target system components that are not needed to distinguish the optimization alternatives.

Finding a matching, ideal performance model for an optimization and target system is hard, or even impossible since such a model may not exist. For the discussions in this paper, a target system consists of a compiler with a specific optimization level, an operating system, and a target architecture with a multi–level memory hierarchy. The difficulty of source–level performance prediction is a result of the different sets of optimizations performed by the target system compilers, and the different advanced features of the target operating system and the target architecture. However, it is important to note that advanced optimization transformations used by the target compiler may actually make the performance model easier. For instance, if memory optimization transformations succeed in dramatically improving data locality, the performance model may not have to distinguish cache misses and hits, i.e., it may assume that all data references have the same cost.

This paper discusses a new framework that supports qualitative performance prediction by identifying the performance factors of a target system that need to be modeled in order to achieve the required performance prediction accuracy. The framework is qualitative in the sense that it determines *what* performance factors may explain a particular performance behavior, but not necessarily *how much* of the observed behavior can be attributed to each performance factor. In other words, the discussed framework performs a sensitivity analysis with respect to the different application and architecture performance factors, such as loop order, number of array references, L_1 and L_2 caches, TLB, and load buffer. We are currently investigating how our qualitative framework can be extended to generate quantitative performance models. However, quantitative performance prediction is beyond the scope of this paper.

Figure 1 shows the overall structure of IPERF (Intelligent PERFormance Prediction), a new framework for qualitative performance prediction. To understand the performance behavior of a source code segment on a target system, the user will provide as input to the IPERF framework the source code, a benchmark set of representative data points of measured execution times, and a description of the compiler and machine architecture of the target system. Based on an analysis of the source code and the specification of the target system, the framework will first determine a lattice of performance factors, typically containing only

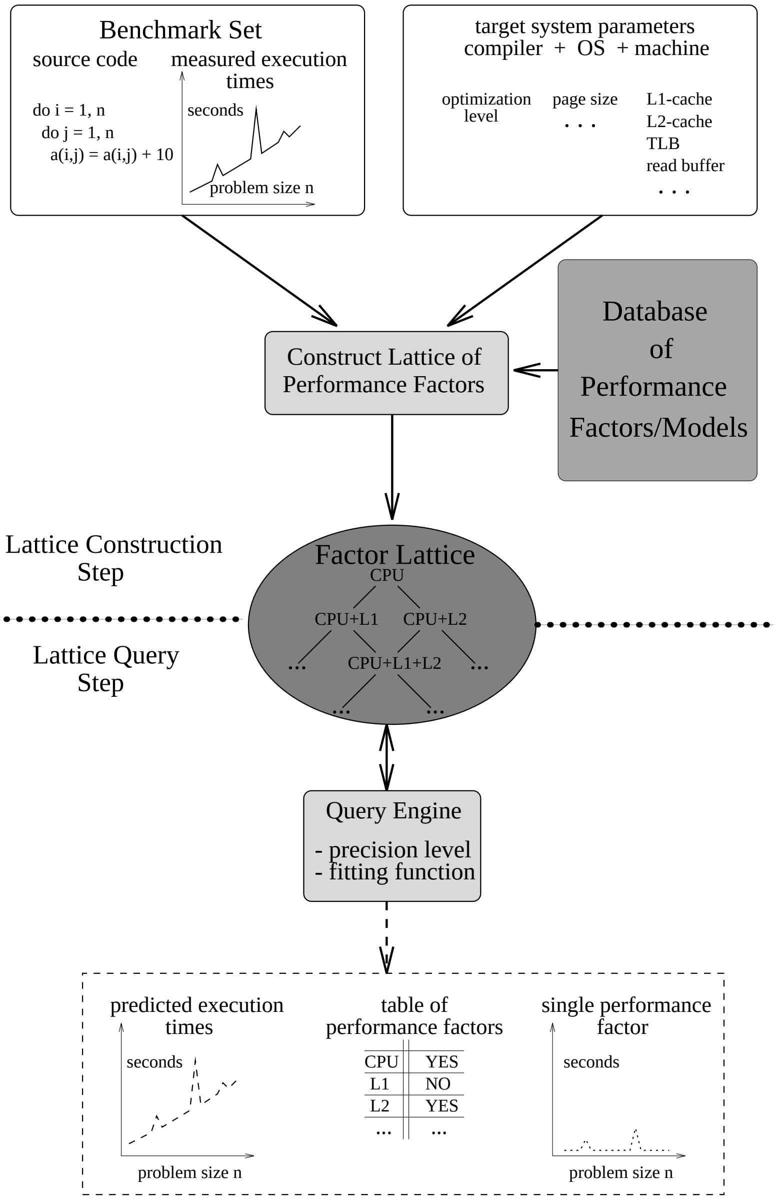


Figure 1: Overview of IPERF , a framework for qualitative performance prediction

a subset of the performance factors represented in the IPERF data base. Once the lattice has been constructed, the precision and cost requirements will direct the query engine to search for the minimal performance model that that approximates the measured behavior of the benchmark set within the specified precision and cost constraints, assuming that such a model can be found, and model fitting characteristic. The resulting model may be presented to the user in different ways, for instance, as a graph of the predicted performance or as a table of the performance factors of the model. In addition, the predicted performance of a single performance factor or subset of performance factors may be displayed.

The current framework supports the least square fit to determine an absolute error distribution, and a weighted least square fit for a relative error distribution. The conceptual difference between a relative and absolute error is illustrated in Figure 2.

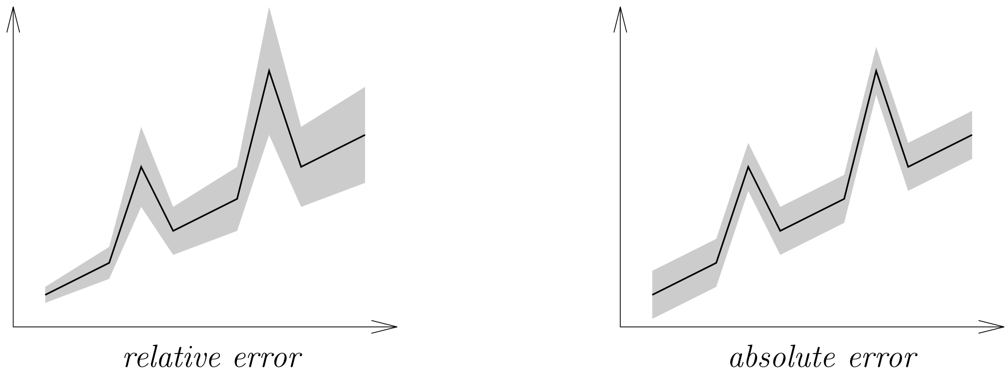


Figure 2: Different precision criteria – shaded area contains data points that satisfy error criterion

The goal of IPERF is to provide an application programmer or compiler writer with support needed to discover and understand complex performance issues and problems for a given target system. IPERF is "intelligent" in the sense that it may use pruning heuristics to construct an appropriate lattice of performance factors, and may consider only a subset of the data base's models for each performance factor in the lattice. Extending IPERF to allow the validation of user supplied partial performance models is beyond the scope of this paper.

The contribution of this paper is a discussion and preliminary evaluation of the design of our IPERF framework. A prototype system has been evaluated, consisting of a combination of automated and hand-simulated steps. The user supplied precision requirement consists of a quantitative error statement (e.g., within 10% of observed performance) and a specification of the requested error as either relative or absolute. Preliminary results show the effectiveness of the new framework. In particular, we were able to identify a performance factor within the operating system based on the fact that no performance model in the generated lattice was able to explain the observed program behavior.

The remainder of the paper is structured as follows. Section 2 discusses the example computational kernel that is used to illustrate the IPERF framework. Section 3 describes the performance factors that IPERF currently considers. In addition, the resulting lattice of performance models is discussed. Section 4 contains experimental results that show the

```
      double precision a(n,n)

      do 10 i=1,n
         do 10 j=1,n
10          a(i,j) = a(i,j) + 1.0
```

Figure 3: Simple Fortran example kernel

generated performance models for different target systems and precision requirements. The paper concludes with a discussion of related and future work in Section 5 and Section 6, respectively.

2 Example

A simple computation kernel (shown in Figure 3) is used to illustrate the design and functionality of the IPERF framework. Although the Fortran kernel is simple, understanding the actual performance behavior on a set of different target systems turns out to be nontrivial. The kernel consists of a doubly nested loop that steps through a double precision array of size $n * n$ with stride n, and increments each element by 1.0.

For the purpose of this paper, we assume that a user needs to determine a performance model that describes the performance behavior of the example kernel for different array sizes n on a specific target architecture with a specific prediction accuracy requirement. In addition, the user wants to understand the cost/precision tradeoffs of different performance models. The current IPERF framework expects as input a set of measured execution time data points as a function of a single parameter, in our example case the array size represented by variable n. In addition, the source code of the measured kernel, a specification of the target system used, and the performance prediction requirements are input to the tool. It is the responsibility of the user to provide a data set that exhibits the performance behavior that the user is trying to understand. Therefore, the choice of such a "representative" data set is outside the scope of IPERF .

In order to illustrate the IPERF framework, we measured the execution times of the example kernel for the set of data points $\{n|\ n = 8i, 1 \leq i \leq 100\}$ on twelve different target systems. For the experiments, a target system consisted of a *(target compiler / target machine)* pair, where the compiler was SUN's SC (SparcCompiler) version 4.0 using three different optimization levels (`-none`, `-O3`, and `-fast -depend`), and one out of four possible SUN SPARC–based target machines (SuperSPARC–I and II, UltraSPARC–I and II). In all cases, the operating system was Solaris 2.5. The diagrams of the resulting measured execution times are shown in Figure 4.

As can be seen in these diagrams, the performance behavior is significantly different for most of the target systems. In particular, IPERF will be used to explain the *performance*

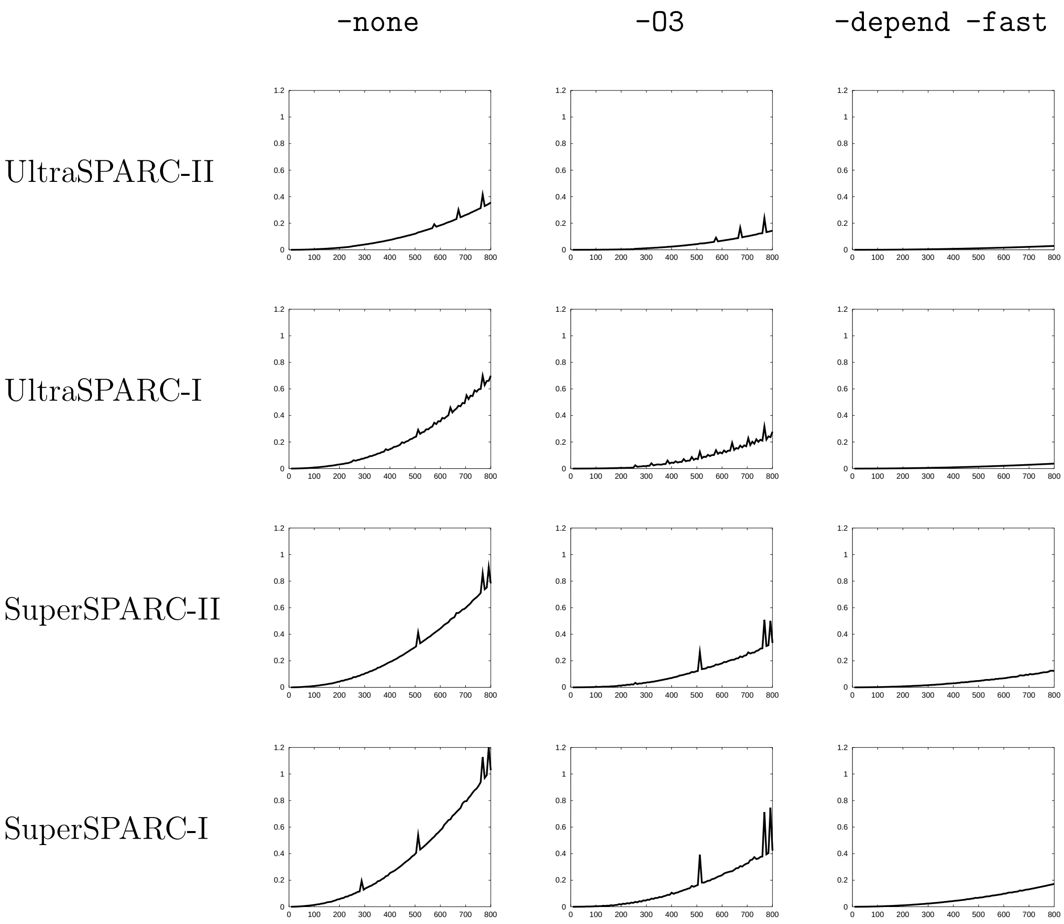


Figure 4: The timing results on twelve different target systems

peaks at different problem sizes n for the target compilers using the `-none` and `-O3` optimization levels.

The exact description of the method used to obtain the measured performance numbers can be found in Appendix A.4. The appendix also contains a discussion of the optimization levels of SUN's SC 4.0 in Appendix A.3. However, the current IPERF prototype system does not consider the characteristics of the target compiler in any of its performance models. The description of the performed transformations at the different optimization levels are provided in order to allow a better interpretation of the observed performance behavior. Including compiler parameters into the IPERF framework is currently under investigation. However, some characteristics of the target machines are considered in the models generated by IPERF . These characteristics are shown in Figure 5.

In general, it is getting harder to model (explain, or debug) the performance of a program when the program becomes larger and more complicated. The motivation for IPERF is that: instead of debugging the performance by observations, IPERF provides an *automatic* way

	UltraSPARC-x		SuperSPARC-x	
	Ultra-II	**Ultra-I**	**Super-II**	**Super-I**
TLB	D:64/8KB/64		D:64/4KB/64	U:64/4KB/64
L1-cache	D:512/32B:16B/1 non-blocking		D:128/32B/4 blocking	
	virtually indexed and physically tagged			
L2-cache	U:32768/64B/1	U:8192/64B/1	U:8192/128B:32B/1	
	blocking			
	physically indexed and physically tagged			

Cache Spec = *Type*:*Sets*/*BlockSize*:*SubBlockSize*/*Associativity*
Type: D (data), U (unified)
In TLB, *BlockSize* indicates page size

Figure 5: A partial description of four SPARC-based machines.

to identify (or suggest) some of the most important performance factors which should be looked at. Applications can range from performance debugging in the scientific computing community to the profitability modeling in the compiler optimization community.

3 Framework

The IPERF framework takes as input a target system specification, a benchmark set of programs with their actual performance measurements, and the precision and cost requirements. The framework will determine a lattice of performance prediction models, based on the models available in data base, and the precision and cost requirements will direct the query engine to search for the minimal performance model in the constructed model lattice that satisfies the requested precision level.

The framework consists of the following major steps:

Step 1: Take the source code, the description of the target system, and the program's performance measurement on the target system (Section 2).

Step 2: Select a subset of possible performance factors and their models from the data base, based on an analysis of the program on the target system (Section 3.1 and 3.2).

Step 3: Search the lattice of performance factors and their models, and evaluate each resulting performance model by comparing predicted values against measured data in the benchmark set. (Section 3.3).

Step 4: Report single or set of performance factors and their models that best fit the user requested accuracy/cost trade-off. A failure is reported if no such models can be found (Section 4).

This section will detail **Steps 2-3**, i.e., how the set of performance factors is selected, the factor lattice is constructed, and the search engine is directed by the evaluation of each possible model.

3.1 Performance Factors

The IPERF framework is based on the assumption that the observed program performance, i.e., its execution time, can be attributed to different aspects of the target system. Each such aspect is called a *performance factor*. There are compiler, operating system, and machine architecture performance factors. The framework uses a data base of performance factors and models for various target systems. Each performance factor in the data base may contain several models with different accuracy/cost trade-offs.

Performance factors are classified as either *compositional* or *numerical*. Models for compositional factors map program representations into other program representations, possibly lowering the level of program abstraction. In contrast, numerical models map program representations into numerical values. Examples of compositional performance factors are compiler optimization passes that perform program transformations such as loop interchange, register allocation, or instruction scheduling. Numerical performance factors include cost models for computation, TLB misses or hits, and the target machine's memory hierarchy.

The IPERF framework assumes that all compiler models are compositional, and all operating system and machine architecture models are numerical. In addition, the framework assumes that the set of all potential performance factors are known for a given target system, including all performed target compiler optimizations and their relative orders, and operating system and target machine characteristics such as page size, cache sizes, cache architecture, number of functional units etc. The final generated model may not consider all performance factors.

Performance factors can be of different granularity, and the associated cost models have different precision. The precision is also dependent on the characteristics of the source programs and the target systems. In this paper we only consider three coarse-grain numerical performance factors, computation (ALU), TLB accesses (TLB), and memory access (MEM), for the example program. The characteristics of the example program include

- Only "one" addition in each iteration [1] appears in the source code segment. This simplicity allows us to only consider single uniform operations in iterations, and not to model structural hazards (i.e., two operations compete the same resource and thus stalled) for operations of different latencies. (ALU)
- Only one array reference appears in the source code segment. This simplicity allows us to only consider locality (i.e., a memory block loaded into the cache are reused without

[1] If we don't take into account other operations such as address computation and branch determination.

Models	definition	remarks
$\mathbf{T_{ALU}}$	$c_0 + c_1 * n + c_2 * n^2$	n : loop bound, assumes unit cost per iteration
$\mathbf{T_{TLB}}$	$c_3 * \begin{cases} 1 \cdot \frac{n^2}{P} & \text{if } n^2 \leq E \cdot P \\ n \cdot \frac{n^2}{P} & \text{otherwise} \end{cases}$	E : #TLB entries, P : page size [HP96, FST91]
$\mathbf{T_{MEM}}$	$c_4 * T_{L1} + c_5 * T_{L2}$	$L1$: model for L_1 cache, $L2$: model for L_2 cache
$\mathbf{T_{L^b}}$	$\begin{cases} \frac{n^2}{sl} & \text{if } \frac{n}{D} \leq a \\ n^2 & \text{if } \frac{n}{D} \geq a+1 \\ \frac{n}{sl}[n + (sl-1)*(a+1)* \\ \quad (n - a * D)] & \text{otherwise} \end{cases}$ **blocking cache model**	$D = \min\{d > 0 \mid \lfloor \frac{n*d}{l} \rfloor \equiv_{mod(S)} 0\}$ d : reuse distance in iterations a : associativity, S : #sets, sl : subblock size [TFJ94, Wol96]
$\mathbf{T_{L^{nb}}}$	$\begin{cases} n * \frac{n}{D_{L1}} & \text{if } \frac{n}{D_{L1}} > a_{L1} \text{ and } \frac{n}{D_{L2}} \leq a_{L2} \\ 0 & \text{otherwise} \end{cases}$ **nonblocking cache model**	D_{Li} : parameter D for L_i a_{Li} : associativity of L_i

Figure 6: Performance models used in IPERF prototype system.

referring back to memory) of an array, and not to model interferences from other array elements. (no group locality, and no cross-interferences in MEM)

- There is no data dependence across iterations. This simplicity suggests that we only need to model the possible spatial locality since there is no temporal locality. (no self-temporal locality in MEM)

- Only one constant (but user-defined) stride appears in the memory access pattern. This helps us to model the spatial locality more precisely. (constant stride in MEM)

The models used in this paper is listed in Figure 6. Their graphical representations for different target machines are shown in Figure 7.

3.1.1 ALU Costs

This ALU costs correspond to the computational complexity of a program, i.e., the program runs on an ideal unit-cost machine, with no pipeline stalls and memory stalls. To the example program, it simply is

$$\mathbf{T_{ALU}} = c_0 + c_1 * n + c_2 * n^2$$

where c_0, c_1, and c_2 are constants to be determined.

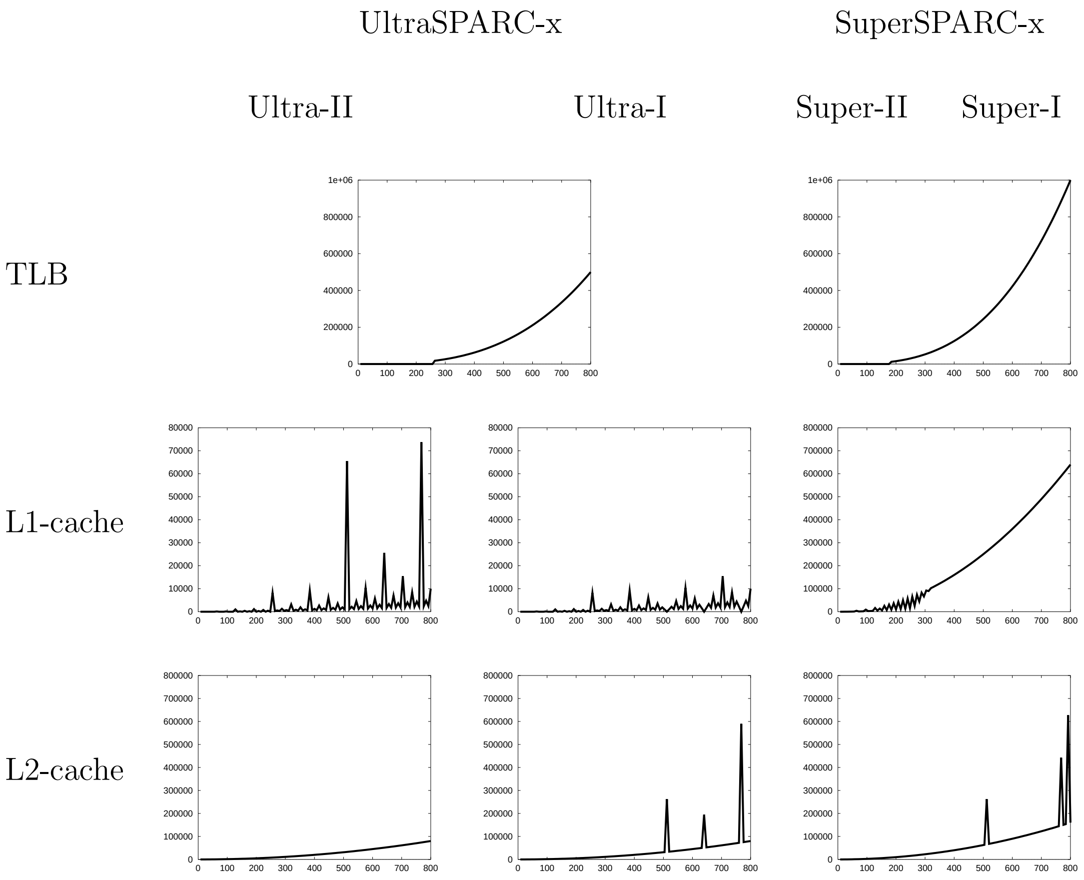


Figure 7: The performance factors of the four target machines.

3.1.2 TLB Costs

TLB is a dedicated cache which stores partial page table to speed up the virtual-to-physical address translation. Since each TLB miss causes the processor trapped to find out the translation, the number of TLB misses is usually used as the TLB costs. TLB is usually fully associative, and therefore the TLB misses are really capacity misses.

Suppose a TLB consists of E entries, and the page size is P (i.e., TLB = $E/P/E$). Furthermore, we assume that $n \leq P$. The capacity misses is estimated by checking whether the whole array (of size n^2) can fit into TLB [2] or not. If not, LRU replacement policy induces approximately the same amount of TLB misses for every row sweep of the array. And the total misses are the multiplication of number of rows (n) and TLB misses per row (n^2/P).

$$\mathbf{T_{TLB}} \approx \begin{cases} 1 \cdot \frac{n^2}{P} & \text{if } n^2 \leq E \cdot P \\ n \cdot \frac{n^2}{P} & \text{otherwise} \end{cases}$$

The idea is not new. It identifies the *overflow loop* [Por89], i.e., the innermost loop that causes the cache to overflow, and gets an estimate of misses by multiplying the misses for the loop nest within the overflow loop by the product of the number of iterations of the overflow loop and its containing loops [FST91]. How to identify such an overflow loop has been one of the research topics [FST91, Pug94, Cla96]. See Appendix A.1 for further elaboration.

3.1.3 Memory Hierarchy Costs

Multi-level memory hierarchy in general complicates the cost model. The memory access stream at one level of memory hierarchy depends on the misses at the previous level [3]. In other words, memory component at one level serves as the filter of the memory access stream for the next level. It is this filtering feature which makes it much harder to derive an analytical formula for the memory hierarchy costs. Fortunately, if the *inclusion property* [BW88] is maintained in the hierarchy, the memory access stream at any level can be considered as exactly the same. For the case of two-level cache with inclusion property, the cache hierarchy costs can be modeled as

$$\mathbf{T_{CACHE}}(M) = c_4 * \mathbf{T_{L1}}(M) + c_5 * \mathbf{T_{L2}}(M)$$

where M is the memory access stream generated by the CPU, and $\mathbf{T_{L1}}$ and $\mathbf{T_{L2}}$ are the costs of L1- and L2-caches, respectively.

How to compute $\mathbf{T_L}$ will depend on the type of the cache at level L, i.e., blocking or non-blocking. The following two sections will discuss our models to blocking and non-blocking caches. In order to treat each level separately, in our model it is implicitly assumed that *all data requested at the current level will reside at the next level*. Furthermore, we ignore the possibility that the operating system might introduce extra swapping costs [4]. In other

[2] The total capacity of TLB is defined as $E \cdot P$.

[3] One level closer to the CPU.

[4] Some data reside on disk. And, unfortunately, these costs are hard to model since they depend on the workload of the system.

words, we assume that all data reside in the main memory, and get the model for the memory hierarchy costs as

$$\mathbf{T_{MEM}} = \mathbf{T_{CACHE}}$$

3.1.4 Blocking Cache Costs

A cache is *blocking* if it can sustain at most one miss and the subsequent accesses (maybe to different cache blocks) have to wait until this outstanding miss is resolved. As a result, the number of cache misses is used as the blocking cache costs. For our example program, the costs on a $(S/l : sl/a)$ cache are

$$\mathbf{T^b} = \begin{cases} \frac{n^2}{sl} & \text{if } \frac{n}{D} \leq a \\ n^2 & \text{if } \frac{n}{D} \geq a+1 \\ \frac{n}{sl}[n + (sl-1) * (a+1) * (n - a \cdot D)] & \text{otherwise} \end{cases}$$

where D is defined as

$$D = \min\{d > 0 | \left\lfloor \frac{n \cdot d}{l} \right\rfloor \equiv 0 \pmod{S}\}$$

The value of D indicates the minimal number of iterations that the same cache line will be reused for different array elements. If a cache line is mapped by the array elements $(\frac{n}{D})$ more than its capacity (i.e., its associativity a), then every access of the line will induce one capacity miss. See Appendix A.2 for more details about this cost model.

3.1.5 Non-Blocking Cache Costs

In contrast to blocking cache, a (load) miss in a non-blocking cache does not block subsequent cache accesses [5], and thus reduces stalls on misses and allows out-of-order completion [6] [HP96]. There is one exception: load referring to the cache block being fetched will be blocked until the fetch is done. Otherwise, a non-blocking cache can sustain one (load) request at a time (i.e., the cache is *pipelined*). If L denotes the load latency, then a simple model for such a "traffic jam" (illustrated in Figure 8) can be

$$\mathbf{T^{nb}} = \max(L - D, 0) \cdot \frac{n}{D} \approx \frac{n}{D}$$

where L is the load latency and D is defined above.

Intuitively, a request to access the same cache line is generated at every D iterations. If between the two reuses the request cannot be completed $(D < L)$, the subsequent reuse has to be blocked and an extra $L - D$ delay is introduced. Since there are $\frac{n}{D}$ such reuses (for a cache line) and the delay is accumulated, $(L - D) \cdot \frac{n}{D}$ gives the total delay. Note that load latency L does not refer to the hardware timing. It is relative to the request rate

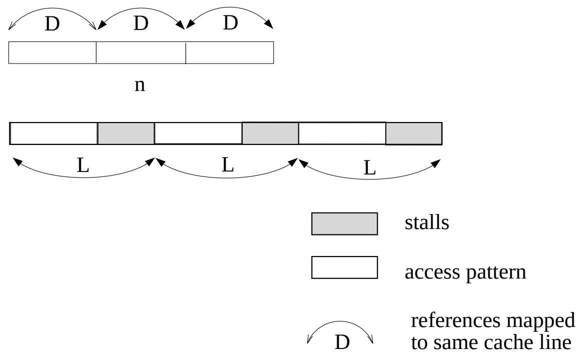


Figure 8: Modeling the stalls of the non-blocking cache.

D. Therefore, it becomes hard to determine L and an approximation is used instead in this paper.

Finally, we have to adjust the formula slightly to conform to the assumption that all data requested at current level reside in the next level (see Section 3.1.3). For our example program on a two-level cache with the first level non-blocking and the second level blocking, the non-blocking cache costs are estimated as

$$\mathbf{T^{nb}} = \begin{cases} n \cdot \frac{n}{D_{L1}} & \text{if } \frac{n}{D_{L1}} > a_{L1} \text{ and } \frac{n}{D_{L2}} \leq a_{L2} \\ 0 & \text{otherwise} \end{cases}$$

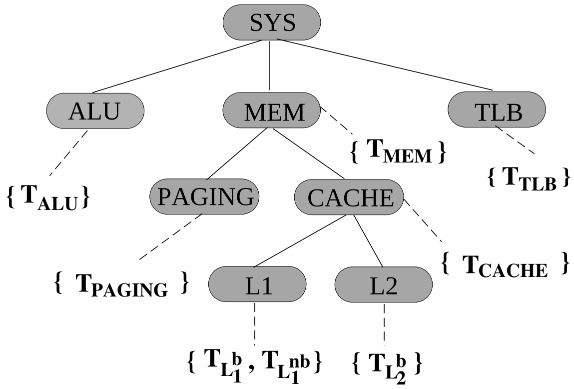
3.2 Factor Lattice

In IPERF , a complete performance prediction model is defined as a linear combination of models from the data base, each of which describes a different performance factor. The data base is organized as a tree, where nodes represent performance factors, and edges represent a refinement relation between factors. Each performance factor in the data base is associated with a set of models. Figure 9(a) shows a sample IPERF data base.

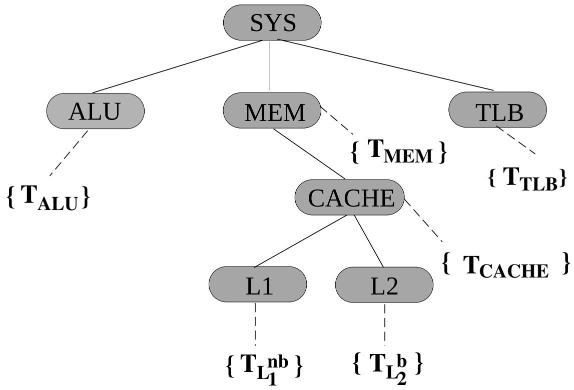
Based on the analysis of the program and the specification of the target system, the factor tree in the data base is pruned in such a way that only possible performance factors and models are kept for search. This pruning process is called *pre-selection* in IPERF . For example, if the target machine is Ultra-I, then we know from its specification (Figure 5) that it has two-level cache and L1-cache is non-blocking. Therefore, the model $\mathbf{T^{b}_{L1}}$ associated with performance factor L1 can be pruned since it describes a blocking L1-cache. Similarly, the analysis of the example program may indicate that there is no page faults, and therefore performance factor PAGING is pruned as well. The pruned factor tree is shown in Figure 9(b).

[5]Different implementations give different limitations of which types of subsequent accesses can proceed, e.g., "hit under miss".

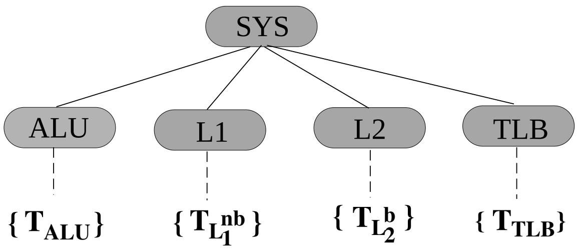
[6]In UltraSPARC-x, all loaded data are enforced to be returned in order.



(a) The structure of a sample IPERF data base



(b) The factor tree during the pruning process



(c) The final pruned factor tree

A selected (not-pruned)performance factor can either be represented by a model directly associated with its node, or by a linear combination of models for its children in the factor tree. If a performance factor does not have any direct model associated, it is considered pruned since it will be represented by its children factors in the tree. For example, assume the current IPERF data base does not have any model directly associated with performance factors MEM and CACHE. Not only the models $\mathbf{T_{MEM}}$ and $\mathbf{T_{CACHE}}$ but also the factors MEM and CACHE are pruned. The final pruned factor tree is shown in Figure 9(c).

3.3 Model Search

Once the set of performance factors and models are selected, the next step is to enumerate all possible complete models and evaluate the quality of them. Recall that a complete performance model is defined as a linear combination of the models from the pre-selected models, each of which represents one pre-selected performance factor. The complete model may not consider all pre-selected performance factors. A particular enumeration of the complete models from the pruned tree Figure 9(c) is shown in Figure 9.

no.	Models	cost
1	$c_0 * \mathbf{T_{ALU}}$	1
2	$c_0 * \mathbf{T_{ALU}} + c_1 * \mathbf{T_{TLB}}$	2
3	$c_0 * \mathbf{T_{ALU}} + c_1 * \mathbf{T_{L_1^{nb}}}$	2
4	$c_0 * \mathbf{T_{ALU}} + c_1 * \mathbf{T_{L_2^{b}}}$	2
5	$c_0 * \mathbf{T_{ALU}} + c_1 * \mathbf{T_{L_1^{nb}}} + c_2 * \mathbf{T_{L_2^{b}}}$	3
6	$c_0 * \mathbf{T_{ALU}} + c_1 * \mathbf{T_{TLB}} + c_2 * \mathbf{T_{L_1^{nb}}} + c_3 * \mathbf{T_{L_2^{b}}}$	4

Figure 9: All possible generated models for factor tree in Figure 9(c).

The coefficients c_i with each complete model are determined through *fitting*: find the best fit against the measured benchmark set data, with respect to the user's precision requirement. Coefficients here are not supposed to interpreted as the absolute timing costs, but the relative importance among performance factors. In the current implementation of IPERF , two fitting methods are provided. Each method is used for a particular context of user's request. They are ordinary least-square fitting (OLS) and weighted least-square fitting (WLS). OLS concerns more about the absolute errors, while WLS deals with the relative errors. Figure 10 gives their definitions and using contexts.

The complete models are evaluated according to the user's requirement. For example, the requirement `Prob(AE` $\leq$ `a)` $\geq$ `b%` states that

> Is it true that this model has more than `b%` (say 85%) probability to predict the absolute execution time within `a` (say 5ms)?

Depending on the user's requirement, the highest-precision model or the minimal-cost model will be reported. The details for these requirements are discussed in Section 4. Note that

User's Requirement	Fitting Method	Definition
Prob(AE $\leq$ a) $\geq$ b%	OLS	$\min \sum_I [AE(I)]^2$
Prob(RE $\leq$ a) $\geq$ b%	WLS	$\min \sum_I [RE(I)]^2$

I = instance of the training set
AE = absolute error
RE = relative error
OLS = ordinary least-squares fitting
WLS = weighted least-squares fitting

Figure 10: The fitting methods used in this paper.

for some cases there do not exist such a model. There are three possible reasons for such a situation: (1) The measured benchmark set contains a lot of noises. (2) The user's precision request is too strong, while the models in the IPERF data base are too approximate. (3) Some significant performance factors are missing. Section 4 shows that the non-existed-model situation helps the process of the *performance debugging*.

4 Experiments

In this paper four sets of experiments are performed, demonstrated, and discussed:

Experiment 1: Varying precision requests – different precision requests result in different performance prediction models on a single target system.

Experiment 2: Varying target systems – different target systems lead to different performance prediction models on a single precision request.

Experiment 3: Varying compiler optimization levels – different versions of a program may have the same performance prediction model for a single compiler optimization level.

Experiment 4: Varying operating system settings – different operation system settings may induce different behaviors of the same program on a single target machine.

4.1 Experiment 1

By fixing a single target system (`-O3`,Ultra-I), with the 100 data points shown in Figure 4, this experiment chooses from the set of performance factors $\{\mathrm{CPU}, \mathrm{TLB}, \mathrm{L}_1^b, \mathrm{L}_1^{nb}, \mathrm{L}_2^b, \mathrm{L}_2^{nb}\}$, and uses least-square fitting to determine the coefficients. Figure 11 shows a partial list of models in this experiment and their quality. It is interpreted, for example, as: Model 5 can

Model	AE $\leq$ 0.005	AE $\leq$ 0.01	AE $\leq$ 0.015
1. CPU	58	74	87
2. CPU+TLB	57	77	84
3. CPU+L_1^{nb}	64	83	96
4. CPU+L_2^b	60	79	93
5. CPU+L_1^{nb}+L_2^b	81	85	89
6. CPU+TLB+L_1^{nb}+L_2^b	88	98	99
7. CPU+TLB+L_1^b+L_2^b	67	79	94
8. CPU+TLB+L_2^b	60	82	92
9. TLB+L_1^{nb}+L_2^b	70	92	97

Figure 11: The Target System: (-O3,UltraSPARC-I)

predict 81% (81 out of 100) of data points within absolute error 0.005 for each case. The different definitions of the *best* model will make IPERF report different models. In this paper, we focus on two definitions:

(**Highest-precision**) The best model is always the one with the highest percentage of satisfying prediction. If there are more than one "best" model, the cheapest model is selected.

(**Cheapest**) The best model is the one with the fewest factors involved while within the (user-specified) percentage of satisfying prediction.

4.1.1 Different precision requests lead to different performance models on a target system

As shown in Figure 12(a), if the highest-precision is the concern, then Model 6 will always be reported. However, there is one exception. When the user requests a model that 90% of prediction should be within 0.005 absolute error, IPERF *cannot* find any model. It might indicate that either the performance factors are not detailed, or the data points provided by the user are not precise enough. Section 4.4 gives an example when this kind of situation happens and how it helps us to dig out the unexpected factors. In any case, the user gets the valuable feedback from IPERF all the time.

If the cheapest model is what we want, Figure 12(b) shows the models reported by IPERF . Clearly, different models are chosen in different precision requests. This example illustrates one motivation behind IPERF : different contexts for using performance models define differently what the best model is and may result in different performance prediction models. The model with the highest precision may not always be welcomed, because, for instance, it is too costly.

	90%	80%	70%
0.005	-	6	6
0.01	6	6	6
0.015	6	6	6

(a) With highest precision

	90%	80%	70%
0.005	-	5	5
0.01	6	3	1
0.015	3	1	1

(b) With fewest factors

Figure 12: Model Selection

4.1.2 The sensitivity of the performance factors to the overall performance

Besides returning the "best" performance model, IPERF also provides a way to do sensitive analysis of performance factors, i.e., help determining the relative contribution of each performance factor to the overall performance. This hot-spot detection phase is needed in the so-called *performance debugging*. Note that, though IPERF provides hot spot identification, these spots might not be real in practice. This is due to the nature of *fitting* in IPERF . Nevertheless, it provides *suggestions* to the performance debugger of which performance factors should be checked first.

Take Figure 11 for an example. The list of models suggests at least the following:

- CPU is a very important performance factor. (compare Model 6 and 9)
- TLB is not as important as CPU (compare Model 1 and 2).
- Memory hierarchy cost is also important (compare Model 1, 2, and 5).
- It is even more important to select the *right* memory cost model (compare Model 6 and 7). The traditional way of counting cache misses as the cost does not predict well in the advanced architectures with non-blocking cache.

Let's go further to examine the argument "why TLB is not as important as CPU". Intuitively, UltraSPARC-I uses physical caches, and thus TLB costs should be taken into account. But if we examine Figure 7 for TLB costs carefully, we will find that the curve of TLB cost is the same as the curve of CPU cost, when n is sufficiently large. A reasonable explanation might be that, DTLB is so small that every TLB access will induce a miss. This example brings an important nature of *fitting* that coefficients may not be able to distinguish where a particular performance factor goes. In this case, CPU model represents both CPU cost and TLB cost, and thus make TLB cost model "useless".

4.2 Experiment 2

This experiment runs over the 12 different target systems, and focuses on the highest-precision model for each target system. Figure 13 shows the results of this experiment.

target system		performance models						
Arch	Opt	CPU	TLB	L_1^b	L_2^b	L_1^{nb}	L_2^{nb}	AE ≤ 0.005
Ultra-II	-none	$\surd$				$\surd$		90
	-O	$\surd$				$\surd$		90
	-fast	$\surd$						100
Ultra-I	-none	$\surd$	$\surd$		$\surd$	$\surd$		88
	-O	$\surd$	$\surd$		$\surd$	$\surd$		88
	-fast	$\surd$						100
Super-II	-none	$\surd$			$\surd$			84
	-O	$\surd$			$\surd$			78
	-fast	$\surd$						100
Super-I	-none	$\surd$	$\surd$		$\surd$			76
	-O	$\surd$	$\surd$		$\surd$			78
	-fast	$\surd$						100

Figure 13: With highest precision

4.2.1 Different target systems lead to different performance models on a precision request

In contrast to Section 4.1.1, target system is another significant factor for choosing the best performance model. According to Figure 13, we can conclude the following for the example program:

- No matter on which target machine, optimization level `-depend -fast` always smoothes out the peaks and makes the performance model look like a simple CPU model.
- No matter in optimization level `-none` or `-O3`, the same target machine always has the same best performance model.
- When the optimization level is either `-none` or `-O3`, Ultra-II has a different best performance model from the Ultra-I, while the best models for Super-I and Super-II are the same.

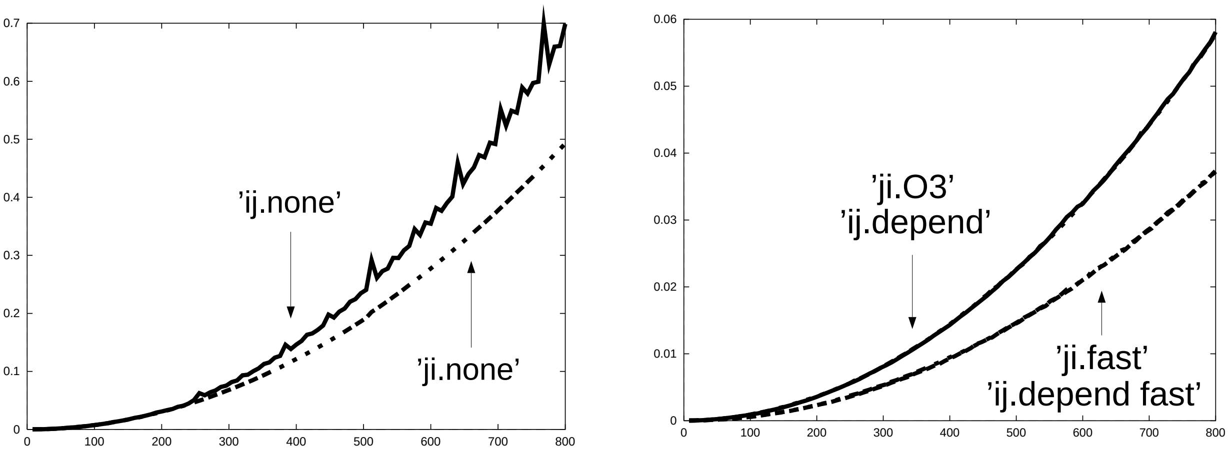
Highly optimized code tends to eliminate (or hides) the cases of heavy performance bottlenecks. This seems to explain why optimization level `-depend -fast` induces the same best performance model, regardless of different target architectures. In fact, Section 4.3 will further explore this by identifying that the loop interchange technique reorders the memory access pattern so that it eliminates the heavy cache conflict misses shown as peaks in the timing results. Note that recent advanced architectures introduce optimization hardwares which will have the same effects as compiler optimizations for the performance.

On the other hand, if the program is not optimized well enough, machine parameters, such as cache sizes, have to be taken into account for the performance model. This seems to explain why target machine determines the best performance model, regardless of the optimization level being `-none` or `-O3`. According to Figure 17, `-O3` does optimizations local to the program loop structure, while `-depend` may rearrange the loop structure. As a result, the same memory pattern, generating heavy conflict misses, is kept and is reflected by the underlying target architecture.

Ultra-I and Ultra-II described in Figure 5 are only different in the size of the L_2 cache. While Ultra-I has a 512KB L_2 cache, Ultra-II has a much larger one (2MB). It suggests that Ultra-II is less sensitive to the cache misses of L_2 cache than Ultra-I (Figure 7 indicates that in Ultra-II there are no peaks in L_2 cache, while in Ultra-I there are three peaks.). This might explain why Ultra-II has a much simpler best performance model than Ultra-I.

4.3 Experiment 3

As Section 4.2.1 describes, peaks are suspected from heavy cache conflict misses, and optimization level `-depend` is conjectured to do the loop interchange for the example program, i.e., from ij-order to ji-order, that the peaks are smoothed out. Using IPERF , we can *verify* this argument by setting another experiment: reverse the loop order of the example program. The measured data points of the two loop orders are shown in Figure 14.



(a) `-none`

(b) `-fast` and/or `-depend`

Figure 14: The timing results for different loop orders in Ultra-I

4.3.1 Different versions of a program may have the same best performance model

From the data points we notice several issues:

- ji-order introduces no peaks, even if no optimization is specified (see curve `ji.none`).
- ij-order with `-depend` is equivalent to ji-order with `-O3`. It indicates that `-depend` no only does loop interchange, but also invokes optimization option `-O3`.
- ij-order with `-depend -fast` is equivalent to ji-order with `-fast`. It gives another evidence for the above argument.

Since there is no peaks in the curve for ji-order, the best performance model will always be the CPU cost model. It seems to suggest that different loop orders of a program may have different best performance models. However, if the optimization option `-depend` is enabled, then the program with ij-order will behave exactly as the program with ji-order. In this case, the two versions of a program have the same best performance model. This experiment gives another evidence for the argument in Section 4.2.1: the target system is involved in determining the best performance prediction model. Even though the program codes are different, highly optimization level may make them the same in terms of performance.

4.4 Experiment 4

Recall that in Figure 7, the data points $n = 512, 640, 768$ are estimated to have much higher L_1 cache misses and there are no peaks in L_2 cache model. However, a careful examination shows that, in reality, peaks happen at $n = 512, 640, 768$. Where do these peaks come from? In terms of IPERF , it says that the set of performance factors used do not capture these peaks. The work by Kessler [KH92] and Lynch [LBF92] suggest that, when the L_2 cache is physical indexed, multi-megabyte and of low associativity, page allocation has an observable effect for the performance. Suspecting that these peaks are from bad page allocation, the experiment is set up: try different page allocation schemes [7]. Figure 15 shows the timing results of two page allocation schemes in Ultra-II.

It is surprised to find out that, with another coloring (`coloring=1`) scheme, the peaks were eliminated. It suggests that page coloring affects the performance of the program and therefore may have to be considered as a separate performance factor.

5 Related Work

In literatures, there are four widely used methodologies for predicting the performance of a program on a target system: description-driven, analysis-driven, execution-driven, and benchmark-driven strategies. Most systems use hybrid of these strategies to balance off the cost and accuracy of the performance prediction models.

Description-driven methodology assumes the performance model is known [vD96, BMSD95]. Variants include system-defined and user-definable models. While the system-defined approach restricts the range of the target systems to be predicted, user-definable approach depends heavily on the expert's knowledge in order to make prediction reasonably accurate.

[7] In Solaris 2.5.1, there is a kernel parameter called *consistent_coloring* which can be tuned to select different page coloring schemes. When it is 0, the conventional uncoloring scheme is used.

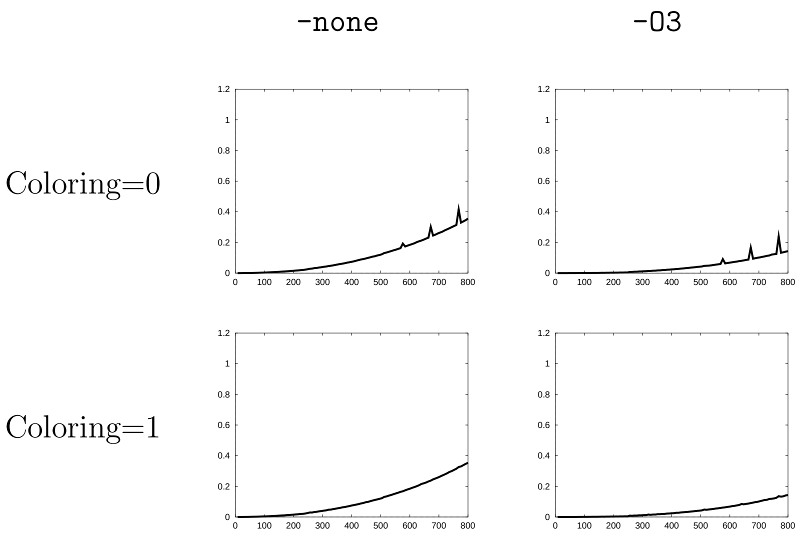


Figure 15: Timings for UltraSPARC-II with two coloring schemes.

Nevertheless, it is argued to be a perfect fit for the fast coarse-grain performance evaluation [vD96].

Analysis-driven methodology replies on the state-of-the-art static analysis [ASU86] to derive the information for performance prediction. Examples include [MW88, Mac94, Fah93, GB94, Wan94, BSM95, CQ97]. It relieves the user from the effort and the expert knowledge to construct the performance model, but the cost and complexity of extracting the information out and developing the satisfactory analysis may restrict its usage in performance prediction.

Execution-driven methodology needs some kind of execution in order to figure out what the performance will be. It is argued that when either the performance itself is non-deterministic or the accuracy of the prediction is the concern, execution-driven strategy is a better choice [dRSBH93]. Systems using execution-driven methodology include [Adi88, dRSBH93, VSV94, Jon95, Tol95, XZS96, SW96].

Benchmark-driven methodology uses data analysis techniques for the measured benchmarks [Jai91] to estimate the predicted value. It is highly dependent on choosing representative benchmarks and quantifying the similarity between them and the predicted. The most popular data analysis techniques include mean value analysis, interpolation, and regression analysis.

The mean value analysis is used in the so-called *microbenchmarking* method [Coh82, SB92]. It estimates the average cost of a primitive operation from a set of microbenchmarks (program kernels). For example, [BFKK91, SB92, dRSBH93, Fah93, KFCK94, BMSD95, XZS96] all use this method to some extent. Some of them [BFKK91, Fah93] even extend to measure say different communication patterns. Microbenchmarking method may suffer from the incompleteness of primitive operations and the non-orthogonality among these operations.

The interpolation method can also be used to estimate the cost of each primitive. While it models the benchmarks implicitly, regression analysis constructs the explicit performance model. The problem of non-orthogonality among operations may be gone, but the computed coefficients cannot be treated directly as the execution cost. It only indicates the *relative* importance among the operations. Examples include [EDF87, Mac94, Ale93, KFCK94, Cro94, Str97]. In our proposed framework, the benchmark-driven methodology, associated with regression analysis, is applied.

Recently, focus has been put on the interaction between different components of a target system and the application [And91, MCF97, MLCH97, ABE+97]. Some researches attempt to model the performance of superscalar processors in conjunction with the exploited instruction parallelism from the source application [Wan94, NS94, BF95, NS97]. Others try to quantify the additional memory access costs, due to the significant amount of cache misses [FST91, WL91, BJWE92, TFJ94, HKN97].

The costs of additional memory accesses are usually estimated in terms of the total amount of cache misses or its miss rate [HP96]. Recent advances introduces the non-blocking cache [Kro81, SD91] as a way to reduce the penalty associated with each cache miss. While the performance of the non-blocking cache is studied [CB92, FJ94, Bal94, WO95], we find in the experiments that the new model accounting for the effect of non-blockness of the cache is needed. In the report, a simple model based on the "traffic pressure" is presented and demonstrated to be better suited than the conventional basis of cache miss counts.

It is observed that, when the cache is multimegabyte, physically-indexed and of low associativity, page allocation schemes affects the cache performance significantly. As a result, different schemes for careful page placement are proposed [Kes91, Lyn93, Mur95, BAM+96]. In our experiments in Section 4, page placement policy plays the major role so that the virtual-memory-based performance models are not capable of predicting the performance peaks in the measurement.

6 Conclusions and Future Work

Finding a source-level performance prediction model with the best precision/cost tradeoff is a non-trivial task. Our framework allows the automatic selection of a performance model. The framework determines the best performance model which satisfies the user supplied precision and/or cost requirements. The performance model selection is "intelligent" since it allows (1) the generation of new performance models based on performance factors represented within the framework's model data base, and (2) efficient search techniques to find models with the requested precision/cost tradeoff. Preliminary experimental results for a small computation kernel and a set of twelve target systems indicate the effectiveness of our framework.

More experiments are needed to validate the effectiveness of our proposed framework. In order to allow experimentation for larger program kernels and whole programs, we are planning to implement a fully automatic IPERF prototype system based on the SUIF compiler infrastructure that is currently being extended as part of the NSF/DARPA National Compiler Infrastructure Project [NCI98]. SUIF contains front-ends for different languages such as Fortran, C, C++, and Java. The set of target machines for the proposed system will span across several current and future architectures.

We are currently investigating different model lattice construction and search strategies for compositional and numerical models. In addition, new methods are being developed that allow the validation of existing performance models and the extension of the current techniques to produce quantitative performance models.

References

[ABE+97] Sarita V. Adve, Doug Burger, Rudolf Eigenmann, Alasdair Rawsthorne, Michael D. Smith, Catherine H. Gebotys, Mahmut T. Kandemir, David J. Lilja, Alok N. Choudhary, Jesse Z. Fang, and Pen-Chung Yew. Theme feature: Changing interaction of compiler and architecture. *Computer*, 30(12):51–58, December 1997.

[Adi88] Ashok K. Adiga. *Performance Modelling of Parallel Computations.* PhD thesis, University of Texas, Austin, 1988.

[Ale93] Theresa Alexander. Performance prediction for loop restructuring optimization. Master's thesis, Oregon Graduate Institute of Science and Technology, July 1993.

[And91] Thomas Edward Anderson. *Operating System Support for High Performance Multiprocessing.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 1991. UW-CSE-91-08-10.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[Bal94] Magnus Balldin. A SPARC Processor Extended with Non-Blocking Loads - Implementation and Evaluation. Master's thesis, Department of Computer Engineering, Lund University, June 1994.

[BAM+96] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 244–255, Cambridge, Massachusetts, October 1996. ACM Press.

[BF95] James E. Bennett and Michael J. Flynn. Performance factors for superscalar processors. Technical Report CSL-TR-95-661, Stanford University, Computer Systems Laboratory, 1995.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, April 1991.

[BJWE92] François Bodin, William Jalby, Daniel Windheiser, and Christine Eisenbeis. A quantitative algorithm for data locality optimization. In Robert Giegerich and Susan Graham, editors, *Code Generation: Concepts, Tools, Techniques*, pages 119–145. Berlin: Springer Verlag, 1992.

[BMSD95] J. Brehm, M. Madhukar, E. Smirni, and L. Dowdy. PerPreT — A performance prediction tool for massively parallel systems. In *Proc. Computer Performance Evaluation: Modelling Techniques and Tools*, volume 977 of *LNCS*, pages 284–299, Heidelberg, September 1995. Springer-Verlag.

[BSM95] Robert J. Block, Sekhar Sarukkai, and Pankaj Mehra. Automated performance prediction of message-passing parallel programs. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.

[BW88] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In H. J. Siegel, editor, *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992. Cs-TR-92-06-03.

[Cla96] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM International Conference on Supercomputing*. ACM, May 1996.

[Coh82] Jacques Cohen. Computer-assisted microanalysis of programs. *Communications of the ACM*, 25(10):724–733, 1982.

[CQ97] Mark J. Clement and Michael J. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 23(10):1405–1420, 1997.

[Cro94] Mark Edward Crovella. *Performance Prediction and Tuning of Parallel Programs*. PhD thesis, University of Rochester Computer Science Department, August 1994.

[dRSBH93] Jan F. de Ronde, Peter M. A. Sloot, Marcel Beemster, and L. O. Hertzberger. A simulation methodology for the prediction of SPMD programs performance. In A. Verbraeck and E. J. H. Kerckhoffs, editors, *European Simulation Symposium 1993*, pages 24–25, Delft, The Netherlands, October 1993. Society for Computer Simulation International.

[EDF87] Phillip Ein-Dor and Jacob Feldmesser. Attributes of the performance of central processing units: A relative performance prediction model. *Communications of the ACM*, 30(4):308–317, April 1987.

[Fah93] Thmoas Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, University of Vienna, Department of Software Technology and Parallel Systems, October 1993. now published by Kluwer Academic Publishers , Boston USA, ISBN 0-7923-9708-8, 296pp, March, 1996 as Automatic Performance Prediction of Parallel Programs.

[FJ94] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance tradeoffs with non-blocking loads. In *21st Annual Symposium on Computer Architecture*, April 1994.

[FST91] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *1991 Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, 1991.

[GB94] M. Gupta and P. Banerjee. Compile-time estimation of communication costs of programs. *Journal of Programming Languages*, 2(3):191–225, September 1994.

[GMM97] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *1997 ACM International Conference on Supercomputing*, July 1997.

[HKN97] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Predicting the cache miss ratio of loop-nested array references. Technical Report CS-RR-336, Department of Computer Science, University of Warwick, Coventry, UK, December 1997.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, second edition, 1996.

[Jai91] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley, New York, 1991.

[Jon95] Henk Jonkers. *Performance Analysis of Parallel Systems: A Hybrid Approach*. PhD thesis, Delft University of Technology, October 1995.

[Kes91] Richard E. Kessler. *Analysis of Multi-Megabyte Secondary CPU Cache Memories*. PhD thesis, University of Wisconsin, Madison, July 1991. CS-TR-91-1032.

[KFCK94] Youngtae Kim, Mark Fienup, Jeffrey C. Clary, and Suresh C. Kothari. Parametric micro-level performance models for parallel computing. Technical Report TR-94-23, Department of Computer Science, Iowa State University, December 1994.

[KH92] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. *Proc of the 8th Annual Int. Symposium on Computer Architecture*, pages 81–87, June 1981.

[LBF92] W. L Lynch, B. K. Bray, and M. J. Flynn. The effect of page allocation on caches. In Wen mei Hwu, editor, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 222–225, Portland, OR, December 1992. IEEE Computer Society Press.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, Calif., April 1991.

[Lyn93] William L. Lynch. *The Interaction of Virtual Memory and Cache Memory.* PhD thesis, Computer Systems Laboratory, Stanford University, October 1993. CSL-TR-93-587.

[Mac94] Neil B MacDonald. Predicting the execution time of sequential scientific kernels. In C. W. Kessler, editor, *Automatic Parallelization: New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 32–44. Vieweg, 1994. AP'93, TR-93-04.

[MCF97] N. Mitchell, L. Carter, and J. Ferrante. A compiler perspective on architectural evolutions. In *Workshop on Interactions between Compilers and Computer Architectures*, February 1997.

[MLCH97] N. Mitchell, J. Ferrante L. Carter, and K. Hogstedt. Quantifying the multi-level nature of tiling interactions. In *10th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 1997. LNCS-1366.

[Mur95] Thomas J. Murray. *Theoretical and Practical Aspects of Virtual Page Placement for Direct-Mapped Caches.* PhD thesis, Department of Computer Science, Clemson University, May 1995.

[MW88] Dieter Müller-Wichards. Performance estimates for applications: an algebraic framework. *Parallel Computing*, 9(1):77–106, December 1988.

[NCI98] The National Compiler Infrastructure (NCI) project. Overview available online at http://www-suif.stanford.edu/suif/nci/index.html., Co-funded by NSF/DARPA, 1998.

[NS94] Derek B. Noonburg and John P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 52–62, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[NS97] Derek B. Noonburg and John Paul Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of The 3rd International Symposium on High Performance Computer Architecture*, pages 298–309, February 1997.

[Por89] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989. TR-88-93.

[Pug94] William Pugh. Counting solutions to presburger formulas: How and why. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 94.

[SB92] Rafael H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, U.C. Berkeley, February 1992. UCB/CSD-92-684.

[SD91] C. Scheurich and M. Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 11(1):25–36, January 1991.

[Str97] E. Strohmaier. Statistical performance modeling: Case study of the NPB 2.1 results. *(Euro-Par '97) Lecture Notes in Computer Science*, 1300:985–??, 1997. UT-CS-97-354.

[SW96] Jens Simon and Jens-Michael Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 675–688. Springer, August 1996.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271, New York, NY, USA, May 1994. ACM Press.

[Tol95] Sivan A. Toledo. *Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms*. PhD thesis, Massachusetts Institute of Technology, June 1995. MIT-LCS-TR-656.

[vD96] Arie Jan Cornelis van Demund. *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, April 1996.

[VSV94] L. Vuurpijl, T. Schouten, and J. Vytopil. A scalable performance prediction method for parallel neural network simulations. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking. International Conference and Exhibition Proceedings. Vol.1: Applications*, pages 396–401, Berlin, Germany, 1994. Springer-Verlag.

[Wan94] Ko-Yang Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 73–84, New York, NY, USA, June 1994. ACM Press.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ont., June 1991.

[WO95] Kenneth M. Wilson and Kunle Olukotun. High performance cache architectures to support dynamic superscalar microprocessors. Technical Report CSL-TR-95-682, Stanford University, Computer Systems Laboratory, June 1995.

[Wol96] Michael J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Co., 1996.

[XZS96] Zhichen Xu, Xiaodong Zhang, and Lin Sun. Semi-empirical multiprocessor performance predictions. *Journal of Parallel and Distributed Computing*, 39(1):14–28, November 1996.

A Appendix

A.1 Estimate the Number of Capacity Misses

The idea in [FST91] is to compute the number of distinct memory blocks (DL) in a loop nest and to use this number to indicate whether the loop nest overflows the cache capacity or not. In [FST91, Pug94, Cla96], different ways to determine the DL value in a loop nest are proposed. Here, for comparison, the formula for the example program is derived according to the idea of [FST91] [8] (again, assume that $n \leq P$):

$$\mathrm{T}_{TLB} = \begin{cases} n \cdot n & \text{if } \frac{n(n-1)}{P} > E \text{ (j is the overflow loop)} \\ n \cdot \frac{n(n-1)}{P} & \text{if } \frac{n^2}{P} > E \text{ (i is the overflow loop)} \\ \frac{n^2}{P} & \text{otherwise (no overflow)} \end{cases}$$

A.2 Estimate the Number of Cache Misses

Traditionally, the total number of cache misses (or the cache miss rate) is used as an indicator for the memory costs. As a result, there are many ways proposed in the literature to count or approximate this number, especially for loop nests [WL91, LRW91, TFJ94, GMM97, HKN97]. For the source code in Figure 18(a), th ideas of reuse set, reuse factor, and reuse distance can be combined to derive an analytical formula to estimate the cache misses [9].

Assume that the memory access stream consists of a reference pattern $[s \cdot i | 0 \leq i < n]$ repeated p times, and there are N such patterns [10]. If it runs on a cache of $D : S/l/a$ [11], then the number of cache misses can be estimated as

$$\mathrm{CM} = N \cdot \begin{cases} n & \text{if } \frac{n}{D} \leq a \text{ (full reuse of the pattern)} \\ p \cdot n & \text{if } \frac{n}{D} \geq a+1 \text{ (no reuse)} \\ n + (p-1) \cdot (a+1)(n - a \cdot D) & \text{otherwise (partial reuse)} \end{cases}$$

where D is the reuse distance in iterations, i.e. the number of iterations that two references in the pattern map to the same cache block:

$$D = \min\{d > 0 | \left\lfloor \frac{s \cdot d}{l} \right\rfloor \equiv 0 \pmod{S}\}$$

Figure 16 gives an illustration of this formula on a direct-mapped cache ($a = 1$).

As an aside, D can be approximated by dropping off the floor function. The derivation of D might be less computationally expensive.

$$D = \min\{d > 0 | \frac{s \cdot d}{l} \equiv 0 \pmod{S}\} = \frac{l \cdot S}{\gcd(s, l \cdot S)}$$

[8] To simplify the formula, the ceiling function suggested in [FST91] is not used.

[9] In the example program, only self-interference misses are possible, i.e., two elements of an array compete for the same cache block.

[10] s is called the *stride*.

[11] number of sets S, block size l, and associativity a

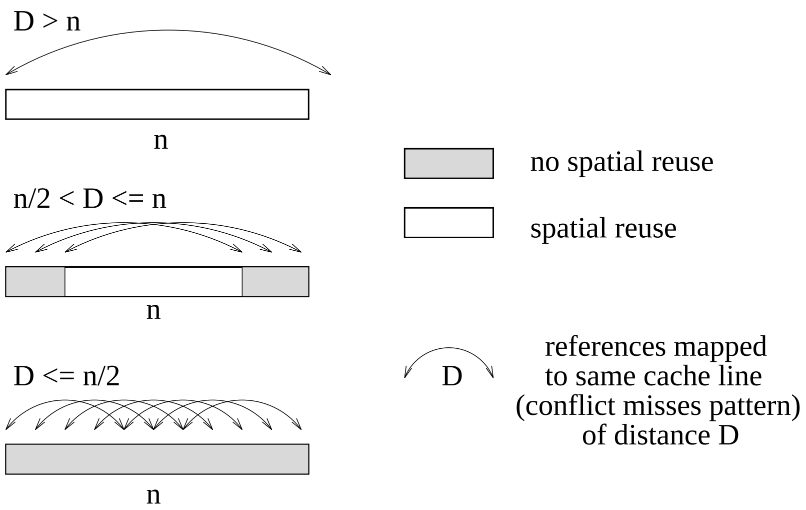


Figure 16: Different situations of a pattern being spatially reused.

A.3 Compiler Options

Each optimization level is a combination of different compiler options available in the compiler. and a target architecture. Figure 17 gives a partial list of compiler options available in Sun's SC version 4.0. Three compiler optimization levels, no optimization (`-none`), `-O3`, and `-depend -fast`, are selected for illustration. Similarly, four SPARC-based target machines (Figure 5) are selected as well, resulting in totally 12 different target systems. Running the source code segment and the driver program/shell on each target system, we collect the benchmark set of measured data points as shown in Figure 4.

A.4 Measurement of Example Kernel Execution Times

In IPERF , the source code segment, a benchmark set of representative data points of measured execution times, and a description of the target system are needed. Figure 18(a) gives the source code segment which we want to model. Since it depends on the array size $n \times n$, data points can be selected by varying different n's. With this in mind, Figures 18(b) and 18(c) are designed as the driver program and shell, respectively, to generate the "representative" data points of measured execution times. Here the representativity means $\{n|\ n = 8i, 1 \leq i \leq 100\}$ Note that, in the driver program, the `dtime()` library function [12] is used to measure the execution time. And in the driver shell, an executable is repeated ten times. The average execution time for a specific n is computed in terms of 80% average, i.e., ignore the smallest and the largest measured times.

The example program kernel has been embedded in a procedure `tested` in order to inhibit optimizations across calls to the timing routines `dtime`. The embedding works, since

[12] According to the manual, the function reports in seconds, and the resolution is to a nanosecond under Solaris 2.x.

Compiler Options [FORTRAN 77 4.0 User's Guide]

-O[n] Optimize for execution time. If not specified, the compiler still executes a single iteration of local common subexpression elimination and live/dead analysis.

-O2 Do basic local and global optimization.

-O3 Besides -O2, this option optimizes references and definitions for external variables.

-depend Analyze loops for inter-iteration data dependencies and do loop restructuring.

-fast Optimize for speed of execution using a selection of options, including -O3 option.

Figure 17: A partial list of compiler options in SPARC Compiler 4.0

no interprocedural optimizations are performed at any used optimization level.

Running the source code segment and the driver program/shell on each of the twelve target systems, we collect the benchmark set of measured data points as shown in Figure 4.

A.5 Complete Data Profile

Figure A.5 gives the list of models constructed and compared against in the experiments, while Figure A.5 and A.5 give all the evaluation results of them in different target systems. Each entry in the table represents s read as: "ρ% of data points are within the error AE $\leq \delta$ in model x".

```
      subroutine tested
      real*8 a(n,n)
      integer i,j

      do 10 i=1,n
         do 20 j=1,n
            a(i,j) = a(i,j) + 1.0
20       continue
10    continue

      return
      end
```

(a) The source code segment (tested.f)

```
      program tester(parameter n)
      real*8 a(n,n)
      real e, t(2)

      e = dtime(t)
      call tested(a,n)
      e = dtime(t)

      write(*,1) n, t(1)
1     format(i12,f20.6)

      end
```

(b) The driver program (tester.f)

```
for n=8,800,8
% f77 -c tester.f(n)
% f77 CompilerOptions -c tested.f
% f77 tester.o tested.o
% do-10-times a.out >> testing.temp
% compute-80%-average testing.temp
endfor
```

(c) The driver shell

Figure 18: The training set

1. CPU
2. CPU+TLB
3. CPU+L_1^{nb}
4. CPU+L_2^{b}
5. CPU+L_1^{nb}+L_2^{b}
6. CPU+TLB+L_1^{nb}+L_2^{b}
7. CPU+TLB+L_1^{b}+L_2^{b}
8. CPU+TLB+L_2^{b}
9. TLB+L_1^{nb}+L_2^{b}

Figure 19: A list of models

	UltraSPARC-I								
OLS	-none			-O3			-depend -fast		
	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%
1.	59/75	89/91	93/92	58/44	74/68	87/72	100/94	100/97	100/98
2.	68/91	86/94	93/94	57/45	77/74	84/81	100/96	100/96	100/97
3.	66/76	89/88	96/92	64/56	83/79	96/85	100/94	100/97	100/98
4.	62/78	89/92	96/92	60/53	79/70	93/73	100/94	100/97	100/98
5.	79/77	98/88	99/92	81/61	85/70	89/76	100/94	100/97	100/98
6.	88/88	98/94	99/94	88/66	98/72	99/77	100/96	100/96	100/97
7.	72/92	90/94	97/95	67/55	79/73	94/78	100/96	100/96	100/97
8.	71/92	88/93	97/94	60/50	82/73	92/78	100/96	100/96	100/97
9.	14/26	24/41	27/51	70/49	92/65	97/72	99/17	100/34	100/43
WLS	-none			-O3			-depend -fast		
1.	53/60	61/98	67/100	39/1	44/10	50/26	100/100	100/100	100/100
2.	74/98	87/100	92/100	68/81	79/90	84/96	100/100	100/100	100/100
3.	52/60	62/98	67/100	38/1	45/11	49/23	100/100	100/100	100/100
4.	51/61	62/99	67/100	38/3	44/12	51/28	100/100	100/100	100/100
5.	51/60	63/99	66/100	38/3	46/11	50/24	100/100	100/100	100/100
6.	86/99	95/100	98/100	72/86	85/94	93/97	100/100	100/100	100/100
7.	75/99	90/100	97/100	69/87	81/94	87/97	100/100	100/100	100/100
8.	78/99	89/100	96/100	70/83	80/93	87/97	100/100	100/100	100/100
9.	21/36	32/56	41/69	74/76	86/89	93/93	94/24	98/41	99/53

	UltraSPARC-II								
OLS	-none			-O3			-depend -fast		
	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%
1.	73/70	97/83	98/91	70/56	97/62	98/68	100/96	100/96	100/97
2.	84/85	94/92	97/93	80/77	92/82	97/87	100/96	100/97	100/98
3.	90/67	94/83	95/90	90/47	94/58	95/66	100/96	100/96	100/97
4.	73/70	97/83	97/91	70/56	97/62	97/68	100/96	100/96	100/97
5.	90/67	94/83	95/90	90/47	94/58	95/66	100/96	100/96	100/97
6.	89/87	94/91	95/92	87/64	93/69	95/88	100/96	100/97	100/98
7.	84/85	94/92	97/93	79/75	92/84	97/86	100/96	100/97	100/98
8.	84/85	94/92	97/93	80/77	92/82	97/87	100/96	100/97	100/98
9.	89/94	94/98	95/99	87/63	93/66	95/93	100/96	100/97	100/98
WLS	-none			-O3			-depend -fast		
1.	63/56	67/97	72/99	48/5	58/20	64/35	100/100	100/100	100/100
2.	97/96	97/98	97/100	97/92	97/96	97/96	100/100	100/100	100/100
3.	62/53	66/97	72/100	49/5	58/20	64/35	100/100	100/100	100/100
4.	63/56	67/97	72/99	48/5	57/20	63/35	100/100	100/100	100/100
5.	62/53	66/97	72/100	49/5	58/20	64/35	100/100	100/100	100/100
6.	95/97	96/99	97/100	95/95	96/96	97/97	100/100	100/100	100/100
7.	97/97	97/98	97/100	97/95	97/96	97/96	100/100	100/100	100/100
8.	97/96	97/98	97/100	97/92	97/96	97/96	100/100	100/100	100/100
9.	96/95	96/98	97/100	95/92	96/94	97/95	100/97	100/98	100/98

Figure 20: Complete data profile

SuperSPARC-I									
OLS	-none			-O3			-depend -fast		
	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%
1.	66/86	79/90	84/91	51/64	71/81	78/86	100/90	100/93	100/94
2.	71/83	83/91	85/92	59/56	76/75	80/77	100/95	100/95	100/96
3.	53/84	74/90	82/92	43/60	64/77	70/82	100/90	100/93	100/94
4.	75/91	90/93	93/93	76/77	83/88	88/91	100/90	100/93	100/94
5.	68/91	87/93	91/93	61/71	80/85	86/89	100/90	100/93	100/94
6.	73/94	86/98	91/98	63/71	80/82	87/86	100/95	100/95	100/96
7.	76/94	90/98	92/98	78/74	84/84	88/88	100/95	100/95	100/96
8.	76/92	90/98	92/98	78/75	84/84	88/88	100/95	100/95	100/96
9.	8/22	13/33	17/45	19/19	28/33	42/49	23/19	46/33	69/42
WLS	-none			-O3			-depend -fast		
1.	50/91	69/98	74/99	49/33	54/81	58/87	100/99	100/100	100/100
2.	77/96	83/98	84/99	63/74	77/96	83/97	100/99	100/100	100/100
3.	51/91	68/98	74/99	50/32	54/81	58/88	100/99	100/100	100/100
4.	48/93	63/99	72/99	47/30	52/82	57/88	100/99	100/100	100/100
5.	48/93	64/99	72/99	48/30	52/82	56/88	100/99	100/100	100/100
6.	74/98	84/99	87/99	61/77	78/97	81/99	100/99	100/100	100/100
7.	76/98	84/99	87/99	57/77	80/97	84/99	100/99	100/100	100/100
8.	76/98	84/99	87/99	56/70	79/97	84/99	100/99	100/100	100/100
9.	11/21	16/37	21/52	25/25	43/47	57/69	28/16	53/33	79/48

SuperSPARC-II									
OLS	-none			-O3			-depend -fast		
	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%	0.005/10%	0.01/20%	0.015/30%
1.	79/90	85/93	86/94	69/68	77/82	82/86	100/94	100/96	100/97
2.	80/87	84/91	86/93	69/59	82/73	83/84	100/94	100/97	100/98
3.	75/88	81/93	86/95	59/69	73/82	80/86	100/94	100/96	100/96
4.	84/88	87/93	95/98	78/76	83/82	94/89	100/93	100/97	100/97
5.	82/86	86/93	95/96	74/78	81/83	92/89	100/93	100/97	100/97
6.	82/87	86/95	97/96	74/72	79/83	90/86	100/94	100/96	100/97
7.	84/90	87/95	98/96	77/76	84/85	93/88	100/95	100/97	100/97
8.	84/89	87/95	98/96	77/75	84/83	93/87	100/94	100/96	100/97
9.	10/21	15/32	20/45	21/18	36/34	53/49	35/20	66/33	99/43
WLS	-none			-O3			-depend -fast		
1.	58/92	63/99	76/100	49/24	54/79	60/87	97/100	100/100	100/100
2.	77/97	85/99	85/100	70/91	80/95	85/98	100/100	100/100	100/100
3.	58/97	63/99	77/100	51/27	54/80	60/87	97/100	100/100	100/100
4.	57/94	64/100	69/100	49/25	53/81	60/87	97/100	100/100	100/100
5.	57/94	64/100	69/100	50/29	54/82	59/88	97/100	100/100	100/100
6.	68/99	84/100	87/100	63/93	80/98	86/99	100/100	100/100	100/100
7.	72/99	84/100	85/100	62/90	80/97	85/99	100/100	100/100	100/100
8.	73/99	84/100	85/100	63/90	81/96	85/99	100/100	100/100	100/100
9.	13/21	20/37	25/53	32/23	55/50	68/71	41/17	80/33	87/51

Figure 21: Complete data profile (continued)