

A Modal Model of Memory^{*}

Nick Mitchell¹, Larry Carter², and Jeanne Ferrante³

¹ IBM T.J. Watson Research Center
nickm@us.ibm.com

² University of California, San Diego and San Diego Supercomputing Center
carter@cs.ucsd.edu

³ University of California, San Diego
ferrante@cs.ucsd.edu

Keywords: performance, model, cache, profiling, modal

^{*} Contact author: Nick Mitchell, who was funded by an Intel Graduate Fellowship, 1999-2000. In addition this work was funded by NSF grant CCR-9808946. Equipment used in this research was supported in part by the UCSD Active Web Project, NSF Research Infrastructure Grant Number 9802219.

Abstract. We consider the problem of automatically guiding program transformations for locality, despite incomplete information due to complicated program structures, changing target architectures, and lack of knowledge of the properties of the input data. Our system, the *modal model of memory*, uses limited static analysis and bounded runtime experimentation to produce performance formulas that can be used to make runtime locality transformation decisions. Static analysis is performed once per program to determine its memory reference properties, using *modes*, a small set of parameterized, kernel reference patterns. Once per architectural system, our system automatically performs a set of experiments to determine a family of kernel performance formulas. The system can use these kernel formulas to synthesize a performance formula for any program’s mode tree. Finally, with program transformations represented as mappings between mode trees, the generated performance formulas can be used to guide transformation decisions.

1 Introduction

We consider the problem of automatically guiding program transformations despite incomplete information. Guidance requires an infrastructure that supports queries of the form, “under what circumstances should I apply this transformation?” [32, 27, 2, 16, 30]. Answering these queries in the face of complicated program structures, unknown target architecture, and lack of knowledge of the input data requires a combined compile-time/runtime solution. In this paper, we present our solution for automatically guiding locality transformations: the *modal model of memory*. Our system combines *limited* static analysis with *bounded* experimentation to take advantage of the *modal* nature of performance.

1.1 Limited Static Analysis

Many compilation strategies estimate the profitability of a transformation with a purely *static* analysis [9, 28, 26, 23, 10, 13], which in many cases can lead to good optimization choices. However, by relying only on static information, the analysis can fail on two counts. First, the underlying mathematical tools, such as integer linear programming, often are restricted to simple program structures. For example, most static techniques cannot cope with indirect memory references patterns, such as $A[B[i]]$, except probabilistically [17]. The shortcomings of the probabilistic technique highlight the second failure of purely static strategies. Every purely static strategy must make *assumptions* about the environment in which a program will run. For example, [17] assumes that the B array is sufficiently long (but the SPECint95 `go` benchmark uses tuples on the order of 10 elements long [22]), whose elements are uniformly distributed (but the NAS integer sort benchmark [1] uses an almost-Gaussian distribution), and distributed over a sufficiently large range, r (for NAS EP benchmark, $r = 10$), and, even if r is known, might assume a threshold $r > t$ above which performs suffers (yet, t clearly depends on the target architecture). Our approach applies just enough static analysis to identify intrinsic memory reference patterns, represented by a tree of parameterized *modes*. Statically unknown mode parameters can be instantiated whenever their values become known.

1.2 Bounded Experimentation

Alternatively, a system can gather information via experimentation with candidate implementations. This, too, can be successful in many cases [4, 3, 31]. For example, in the case of tiling, it could determine the best tile size, given the profiled program’s input data, on a given architecture [3, 31]. However, such information is always relative to the particular input data, underlying architecture, and chosen implementation. A change to a different input, or a different program, or a new architecture, would require a new set of profiling runs. In our system, once per architecture, we perform just enough experimentation to determine the behavior of the system on our small set of parametrized modes. We can then use the resulting family of *kernel* performance formulas (together with information provided at run time) to estimate performance of a program implementation.

1.3 Modal Behavior

Our use of modes is based on the observation that, while performance of an application on a given architecture may be difficult to precisely determine, it is often modal in nature. For example, the execution time per iteration of a loop nest may be a small constant until the the size of cache is exceeded; at this point, the execution time may increase dramatically. The execution time of a loop can also vary with the pattern of memory references: a constant access in a loop may be kept in a register, and so be less costly than a fixed stride memory access. Fixed stride access, however, may exhibit spatial locality in cache, and so in turn be less costly than random access. Our approach, instead of modeling performance curves exactly, is to find the inflection points where performance changes dramatically on a given architecture.

Our system uses a specification of a small set of parametrized memory modes, as described in Section 2, as the basis for its analysis and experimentation. Each code analyzed is represented as a tree of modes (Section 3). Additionally, once per architecture, our system automatically performs experiments to determine the performance behavior of a small number of mode contexts (Section 4). The result is a set of kernel formulas. Now, given the mode tree representing a code, our system can automatically synthesize a performance formula from the kernel formulas. Finally, with transformations represented as mappings between mode trees, the formulas for the transformed and untransformed code can be instantiated at runtime, and a choice made between them (Section 5).

2 Reference Modes and Mode Trees

The central representation of our system is a *parameterized reference mode*. We introduce a set of three parameterized modes that can be combined into *mode trees*. Mode trees capture essential information about the memory reference pattern and locality behavior of a program. In this paper, we do not present the underlying formalism (based on attribute grammars); we refer the reader to [22].

The central idea of locality modes is this: by inspecting a program’s syntax, we can draw a picture of its memory reference pattern. While this static inspection may not determine the details of the picture *precisely* (perhaps we do not know the contents of an array, or the bounds of a loop), nevertheless it provides enough knowledge to allow the system to proceed.

2.1 Three Locality Reference Modes

Let’s start with the example loop nest in Fig. 1, which contains three base array references. Each of the three lines in the body accesses memory differently.

```

1  do  $i = 1, 25, 10$ 
2      $A(3)$ 
3      $B(i)$ 
4      $C(A(i))$ 
5  end

```

Fig. 1. An example with three base references.

Line 2 generates the sequence of references $(3, 3, 3)$ ¹, which is a constant reference pattern. Line 3 generates the sequence $(1, 11, 21)$, a monotonic, fixed-stride pattern. Finally, we cannot determine which pattern line 4 generates precisely; it depends on the values stored in A . Yet we can observe, from the code alone, that the pattern has the possibility of being a non-fixed-stride pattern, unlike the other two patterns.

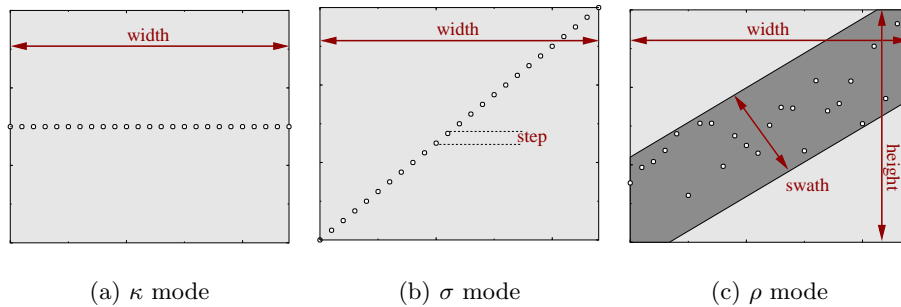


Fig. 2. Visualizing the locality reference modes with memory reference patterns — time flows horizontally, and space flows vertically. Each mode can be annotated with its parameters; for example, the σ mode has two parameters, step and width. More complicated patterns result from the composition of these three modes into mode trees.

¹ Here we use 3 as shorthand denoting the third location of the A array.

Corresponding to these three patterns of references, we define three parameterized reference modes, denoted κ , σ , and ρ . They are visualized in Fig. 2.

constant: κ represents a constant reference pattern, such as (3, 3, 3). Visually, a constant pattern looks like a horizontal line, as shown in Fig. 2(a). This reference pattern has only one distinguishing (and possibly statically unknown) feature: the length of the tuple (i.e., *width* in the picture).

strided: σ represents a pattern which accesses memory with a fixed stride; e.g. (1, 11, 21), as shown in Fig. 2(b). There are two distinguishing (and again, possibly unknown) parameters: the step or stride between successive reference, and the width.

non-monotonic: ρ represents a non-monotonic reference pattern: (5, 12, 4). Visually, a non-monotonic pattern will be somewhere on a spectrum between a diagonal line and random noise; Figure 2(c) shows a case somewhere between these two extremes. A non-monotonic pattern has three possibly unknown parameters: the height, the width, and the point along the randomness spectrum, which we call *swath-width*.

2.2 Mode Trees Place Modes in a Larger Context

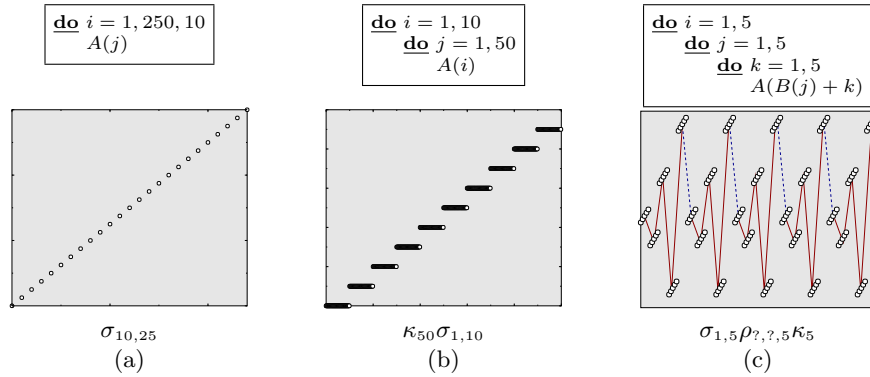


Fig. 3. Three example mode trees. Each example shows a loop nest, a reference pattern picture, and a tree of parameterized *modes*. We write these tree as a strings, so that $\kappa\sigma$ has κ as the child. Some of the parameters may not be statically known; we denote the value of such parameters by question marks (?).

Many interesting memory reference patterns cannot be described by a single mode. However, we can use a *tree* of modes for more complex situations. Consider the example given in Fig. 3(b): a doubly-nested loop with a single array reference, $A(i)$. With respect to the j loop, the memory locations being referenced do not change; this pattern is an instance of κ , the constant mode. With respect to the i loop, the memory locations fall into the σ mode. The reference pattern of this

example is the *composition* of two modes: first κ (because j is the inner loop), then σ . If we draw this relationship as a tree, σ will be the parent of κ . Fig. 3(b) linearizes this tree to a string: $\kappa\sigma$, for notational cleanliness.² Figure 3(c) is an example of a triply-nested loop. Nested loops are instances of the parent-child mode relationship.

We can also have *sibling* mode relationships. This would arise for example in Fig. 1, where a single loop nest contains three array references. We do not discuss this case in this paper. In short, sibling relations require extending (and nominally complicating) the mode tree representation.

3 Lexical Analysis from Codes to Modes

In Sec. 2, we introduced the language of modes: a mode is a class of abstract syntax trees, and a mode tree represents a mode in a larger context. We now briefly describe, mainly by example, how to instantiate a mode tree from a given abstract syntax tree.

First, identify some source expression of interest. For example, say we are interested in the expression $i * X + k$ in a three-deep loop nest, such as in Figure 4(a); say X is some loop-invariant subexpression. From this source expression, we can easily create its corresponding abstract syntax tree (AST), shown in Figure 4(b).

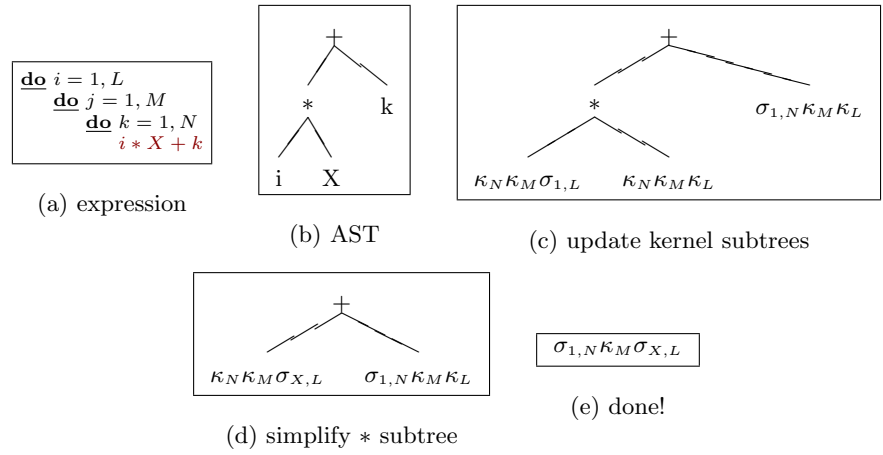


Fig. 4. A lexical analysis example.

From the abstract syntax tree, we next update the “kernel” subtrees. Recall from Sec. 2, that a reference mode is a class of abstract syntax trees. A subtree is a *kernel* subtree of an AST if it belongs to the class of some reference mode [22]. For example, in the AST of Fig. 4(b), the mode σ validates the leaf nodes i and

² Keep in mind that $\kappa\sigma$ denotes a tree whose leaf is κ .

k (because they are induction variables), while the mode κ validates the leaf node X (because it is a loop invariant). Therefore, our example has three kernel subtrees. Now, observe that each kernel subtree occurs on some path of loop nesting. For our example, each occurs in the inner loop, which corresponds to the path $(kloop, jloop, iloop)$, if we list the loops from innermost to outermost. Observe that, with respect to each loop along this path, a kernel subtree in some reference mode M either corresponds to an instance of κ (when the kernel subtree is invariant of that loop) or to an instance of M . This means that, to each kernel subtree, we can write a string of modes. For example, for the kernel subtree i we write $\kappa\kappa\sigma$; for k we write $\sigma\kappa\kappa$; and for X we write $\kappa\kappa\kappa$. Then, instantiate each of the modes appropriately (see [22] for more detail). In our example, we will replace the kernel subtree i , with the mode tree $\kappa_N\kappa_M\sigma_{1,L}$. Figure 4(c) shows the result of updating kernel subtrees. Observe, however, that the resulting tree is not yet a mode tree, because it has expression operations as internal nodes.

The final step applies a series of simplification rules to remove expression operations. For example, the addition a κ to any other tree t behaves identically to t alone; the κ does not alter the sequence of memory locations touched by t . Thus we can correctly replace $(+ \kappa t)$ with t . Multiplying by a κ changes the reference pattern by expanding its height (if before a tree references 100 memory locations, now it will reference $100k$). Applying the latter rule element-wise to the $*$ subtree in Figure 4(c) yields Figure 4(d). Applying the $+$ rule once again finally yields a mode tree, given in Figure 4(e).

4 Performance Semantics from Modes to Performance

Once we have a mode tree, the next step is to determine how this mode tree performs under various circumstances. For example, what are the implications of using an implementation $\sigma_{10^3,10^3}\kappa_{100}$ versus $\sigma_{10^3,10}\kappa_{100}\sigma_{10^4,100}$?³ Our system predicts the performance of mode trees by composing the models for the constituents of a tree. It generates the constituent models from data it collects in a set of experiments. We call this experimentation process *mode scoping*.

Mode scoping determines how the performance of a mode instance m in a mode tree T varies. The performance of m depends not only on its own parameters (such as the swath-width of ρ , or the step of σ), but also on its *context* in T . The remainder of this section describes how:

1. we augment each mode instance to account for contextual effects
2. the system runs experiments to sweep this augmented parameter space
3. the system generates *kernel formulas* which predict the performance of modes-in-context
4. the system predicts the performance of a mode tree by instantiating and composing kernel formulas.

³ Observe that the latter corresponds to the *blocked* version of the former, with a block/tile size of 10. Section 5 discusses transformations.

Our system, driven by specifications from (1), autonomously performs the operations (2) and (3), *once per architecture*. Then, once per candidate implementation, it performs operation (4).

4.1 Contextual Effects

We consider two aspects of the *context* of m in T : the performance of m may depend on, firstly, its position in T , and, secondly, on certain attributes of its neighbors. For example, σ 's first parameter, the step distance, is sometimes an important determinant of performance, but other times not. This distinction, whether step is important, depends on context. To elaborate, we compare three mode trees. The first tree is the singleton $\sigma_{10^2,10^2}$. The second and third trees compose two mode instances: $\sigma_{10^2,10^2}\kappa_{10^2}$ and $\sigma_{10^2,10^2}\sigma_{10^4,10^2}$. Observe that each of these three mode trees is the result of composing a common instance, $\sigma_{10^2,10^2}$, with a variety of other instances. On a 400MHz Mobile Pentium II, the three trees take 7, 3.6, and 54 cycles per iteration, respectively.

To account for the effect of context, we *augment* a mode instance by summaries of its neighbors' attributes and by its position. We accomplish the former via *isolation attributes* and the latter via *position modes*. In summary, to account for contextual effects, we define the notion of a *mode-in-context*. The mode-in-context analog to a mode M in position P , $C_{P,M}$, is:

$$C_{P,M} = \langle P, M, I \rangle$$

where I is the subset of M 's isolation attributes pertinent for P (e.g. child isolation attributes are not pertinent to leaf nodes). We now expand on these two augmentations.

Isolation Attributes To account for the effect of parent-, child-, and sibling-parameters on performance, we augment each mode's parameter set by a set of *isolation attributes*. An isolation attribute encapsulates the following observation: the role a neighbor plays often does not depend on its precise details. Instead, the neighbor effect typically depends on coarse summaries of the surrounding nodes in the tree. For example, we have found that ρ is oblivious to whether it's child is κ_{10^6} versus $\sigma_{1,10^6}$. Yet ρ is sensitive to the width of its children (10^6 in both cases). Hence, we assign ρ an isolation attribute of (`child . width`).⁴ Table 1 shows the isolation attributes that we currently define.

Position Modes We distinguish four position modes based on the following observation. For a mode instance m in mode tree T , the effect of m 's parameters

⁴ Notice that by stating " ρ isolates child-width", we enable *compositional* model generation with a bounded set of experiments. Isolation parameters essentially anonymize the effect of context (i.e. child being ρ doesn't matter); they permit the system to run experiments on these summary parameters, instead of once for every possible combination of child subtrees and parent paths.

mode	parent	child	siblings
κ	width	width	—
σ	reuse, width	width	—
ρ	—	width	—

Table 1. Isolation attributes for the three locality modes for parents, children, and siblings. Currently, we do not model sibling interactions.

and its isolation attributes on performance often varies greatly depending on m 's position in T . We thus define four position modes: *leaf*, *root*, *inner*, and *singleton*. These three correspond to the obvious positions in a tree.

4.2 Mode Scoping

To determine the performance of each mode-in-context, the system runs experiments. We call this experimentation *mode scoping*; it is akin to the well-known problem of parameter sweeping. The goal of a parameter sweep is to discover the relationship of parameter values to the output value, the parameter curve. For a mode-in-context, $C_{P,M}$, mode scoping sweeps over the union of M 's parameters and $C_{P,M}$'s isolation attributes.

Neither exhaustive nor random sweeping strategies suffices. It is infeasible to run a complete sweep of the parameter space, because of its extent. For example, the space for the κ mode contains 10^9 points; a complete sweep on a 600MHz Intel Pentium III would take 30 years. Yet, if performance is piecewise-linear, then the system need not probe every point. Instead, it looks for end points and inflection points. However, a typical planar cut through the parameter space has an inflection point population of around 0.1%. Thus, any random sampling will not prove fruitful.

Our sweeping strategy sweeps a number of planar slices through a mode-in-context's parameter space. The system uses a divide-and-conquer technique to sweep each plane, and a pruning-based approach to pick planes to sweep.⁵ Our current implementation runs 10–20 experiments per plane (out of anywhere from thousands to millions of possible experiments). It chooses approximately 60 planes per dimension (one dimension per mode parameter or isolation attribute) in two passes. The first pass probes end points, uniform-random samples, and \log_{20} - and \log_2 -random samples. The goal of the first pass is to discover possible inflection points. The second pass then runs experiments on combinations of discovered inflection points. Because the first pass may have run more experiments than discovered inflection points, the second pass first prunes the non-inflection points before choosing planes to sweep. The result of mode scoping is a mapping from parameter choices to actual performance for those choices.

⁵ A divide-and-conquer strategy will not discover associativity effects, because the effects of associativity do not vary monotonically with any one parameter. This is a subject of future research.

4.3 Generating Kernel Formulas

After mode scoping a mode-in-context, the system then generates a model which predicts the observed behavior. We call these models *kernel formulas*, because they are symbolic templates. Later, the system will instantiate these kernel formulas for the particulars of a mode tree. Instantiating a kernel formula is the straightforward process of variable substitution. For example, a kernel formula might be $3 + p_1^2 + i_2$ — meaning that this kernel formula is a function of the first mode parameter and the second isolation parameter. To instantiate this kernel formula, simply substitute the actual values for p_1 and i_2 .

Our system, similar to [4], uses linear regression to generate kernel formulas. To handle nonlinearities, the regression uses quadratic cross terms and reciprocals. Furthermore, it *quantizes* performance. That is, instead of generating models which predict the performance of 6 cycles per iteration versus 100 cycles per iteration, the system generates models which predict that performance is in, for example, the lowest or highest quantile. We currently quantize performance into five buckets.⁶ The system now has one kernel formula per mode-in-context.

system	mode-in-context	kernel formula
Pentium	$\langle inner, \kappa \rangle$	$1 - 9.5e - 11p_0 + \frac{0.95}{p_0 i_0} + \frac{0.086}{\sqrt{p_0}}$
PA-RISC	$\langle inner, \kappa \rangle$	$2 + \frac{0.05}{p_0^2} + \frac{0.56}{p_0 i_0} + \frac{0.45}{i_0 i_1}$
Pentium	$\langle leaf, \sigma \rangle$	$2.6 + 5.6 \frac{p_0}{10^4} - 6 \frac{\sqrt{p_0 i_0}}{10^2} - 1.6 \sqrt{p_0} - 5.8 \frac{\sqrt{p_1}}{10^4 i_1} - 5.2 \frac{\sqrt{p_0 p_1}}{10^4}$
Pentium	$\langle root, \rho \rangle$	$0.98 - 8.4 \frac{p_1}{10^7 i_0} - 8.5 \frac{\sqrt{p_2}}{10^4 p_0} - \frac{0.76}{i_0^2} + 3 \frac{\sqrt{p_1}}{10^3 i_0} + \frac{1.2}{\sqrt{i_0}}$
PA-RISC	$\langle root, \rho \rangle$	$2 - 7.5 \frac{p_1}{10^7 i_0} + \frac{233}{p_2 i_0} - \frac{1.1}{i_0} + 0.001 \frac{\sqrt{p_0}}{i_0} + 0.003 \frac{\sqrt{p_1}}{i_0}$

Fig. 5. Some example kernel formulas for two machines: one with a 700MHz Mobile Pentium III, and the other with a 400MHz PA-RISC 8500.

4.4 Evaluating a Mode Tree

Finally, the system evaluates the performance of a mode tree by instantiating and composing kernel formulas. We describe these two tasks briefly. Recall that a mode tree T is a tree whose nodes are mode instances. First, then, for each mode instance $m \in T$, the system computes the mode-in-context for m 's mode. This yields a collection of $C_{P,M}$. Next, the system computes the values for the isolation parameters of each $C_{P,M}$. Both of these steps can be accomplished with simple static analysis: position mode can be observed by an instance's location

⁶ Quantizing performance is critical to generating well-fitting models which generalize. For example, within a plane, we often observe step-wise relationships between performance and parameter values — this is typical behavior in systems with multiple levels of cache. With quantization, the system need not model this curve as a step function, which is very difficult with low-degree polynomials.

in the tree, and isolation attributes, as we have defined them in Sec. 4.1, can also be easily derived. The system now has all the information necessary to instantiate the kernel formulas for each $C_{P,M}$: substitute the actual values for mode parameters and isolation attributes into the kernel formula.

The second step in mode tree evaluation composes the instantiated kernel formulas. One could imagine a variety of composition operators, such as addition, multiplication, and maximum. In [22] we explored several of these operators experimentally. Not too surprisingly, we found that the best choices were maximum for parent-child composition, and addition for sibling composition.⁷

5 Transformations from Modes to (better) Modes

Program optimizations map a mode tree to a set of mode trees. The system is driven by any number of optimization specifications. Table 6 provides a specification for tiling. Each rule gives a template of the mode trees to which the transformation applies (*domain*), and the resultant mode tree (*range*). Notice that the transformation specification gives names to the important mode parameters and optimization parameters (like tile size). The domain template trees use capital letters to name context. In [22], we describe in detail how the a specification describes context, and how our system determines the set of all possible applications of a transformation to a mode tree (e.g. tiling may apply in many ways to a single mode tree; perhaps as many as once per loop in the nest).

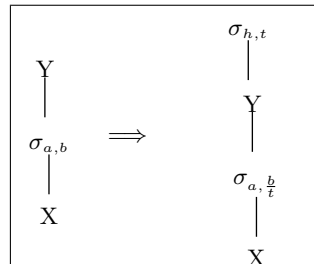


Fig. 6. The transformation specification for tiling: *domain* \implies *range*.

For example, consider loop tiling. Tiling is possibly beneficial whenever a mode tree contains at least one σ , but contains no ρ — as commonly formulated, tiling requires that all references and loop bounds be affine functions of the enclosing loops’ induction variables. To any mode tree which satisfies tiling’s domain criteria, corresponds the set of mode trees which result from tiling one of the loops in the original implementation.

⁷ Recall that our kernel formulas represent cycles per iteration, rather than cycles. Hence, maximum is a good choice for parent-child composition. Had kernel formulas represented cycles, then multiplication would likely have been the best choice.

6 Experiments

We now provide some initial performance numbers. We compare the predicted and actual performance of several implementations. The predicted numbers come from our system, using the performance evaluation strategy described in Sec. 4; the input to the system is the mode tree corresponding to an implementation choice. The actual numbers come from running that actual implementation on a real machine — in this case a 700MHz Mobile Pentium III. In this initial study, we look at loop interchange.

6.1 Loop Interchange

codes	modes
$\underline{\text{do}}\ i = 1, M$ $\quad \underline{\text{do}}\ j = 1, N, s$ $\quad \quad \dots A(B(i) + j) \dots$	$T_1 = \sigma_{s,N} \rho_{?_1, ?_2, M}$
$\underline{\text{do}}\ j = 1, N, s$ $\quad \underline{\text{do}}\ i = 1, M$ $\quad \quad \dots A(B(i) + j) \dots$	$T_2 = \rho_{?_1, ?_2, M} \sigma_{s,N}$

(a)

$S\ ?_1, ?_2$		T_1/T_2	
		pred.	act.
1	10^3	0.84	0.38
1	10^6	0.26	0.64
10	10^6	0.56	0.78
10^3	10^3	1.56	5.68
10^3	10^6	0.94	1
10^3	10^2	1.63	6.12

(b)

Fig. 7. Comparing the two versions of a loop nest. The ratio of T_1 to T_2 is the ratio of predicted performance for that row's parameter value assignment. Notice that both cases have an equal number of memory references.

Figure 7 shows two loop nests, identical in structure except that the loops have been interchanged. Which of the two versions performs better? Phrased in our mode language, we would ask this question: under what conditions will $\sigma\rho$ outperform $\rho\sigma$? The table in Fig. 8(b) shows this comparison for several choices of mode parameters, for both actual runs and for performance as predicted by our system. This table shows T_1/T_2 , which stands for the performance of the implementation represented by mode tree T_1 versus that of T_2 . Thus if $T_1/T_2 < 1$ choose implementation T_1 , if $T_1/T_2 > 1$, choose T_2 , and if $T_1/T_2 = 1$ then the choice is a wash. Observe that for the cases shown in the table, the prediction would always make the correct choice. Figure 8 and shows a similar study with one σ instance versus another.

7 Related Work

Finally, we present research which has inspired our solution. We summarize these works into the following three groups:

codes	modes
$\mathbf{do} \ i = 1, M$ $\mathbf{do} \ j = 1, N$ $\dots A(i * R + j * S) \dots$	$T_1 = \sigma_{S,N} \sigma_{R,M}$
$\mathbf{do} \ j = 1, N$ $\mathbf{do} \ i = 1, M$ $\dots A(i * R + j * S) \dots$	$T_2 = \sigma_{R,M} \sigma_{S,N}$

(a)

		T_1/T_2	
S	R	pred.	act.
1	1	1	1
1	2	0.67	1
1	5	0.52	0.75
1	10	0.46	0.48
1	100	0.43	0.22
1	1000	0.61	0.61

(b)

Fig. 8. Another comparison of two implementations of a simple loop. The ratio of T_1 to T_2 is the ratio of predicted performance for that row’s parameter value assignment; therefore a lower ratio means that the second implementation, T_2 , is a better choice than the first. For every choice of S and R , we chose $N = M = 10^3$.

Combined static-dynamic approaches: Given user-specified performance templates, Brewer [4] derives platform-specific cost models (based on profiling) to guide program variant choice. The FFTW project optimizes FFTs with a combination of static modeling (via dynamic programming) and experimentation to choose the FFT algorithm best suited for an architecture [11]. Gatlin and Carter introduces *architecture cognizance*, a technique which accounts for hard-to-model aspects of the architecture [12]. Lubeck *et al.* [19] use experiments to develop a hierarchical model which predict the contribution of each level of the memory hierarchy to performance.

Adaptive optimizations: Both Saavedra and Park [24] and Diniz and Ri-nard [6] adapt programs to knowledge discovered while the program is running. Voss and Eigenmann describe ADAPT [29], a system which can dynamically generate and select program variants. A related research area is dynamic compilation and program specialization, from its most abstract beginnings by Ershov [8], to more recent work, such as [7, 5, 18, 14].

System scoping: McCalpin’s STREAM benchmark discovers the *machine balance* of an architecture via experimentation [20]. In addition to bandwidth, McVoy’s and Staelin’s lmbench determines a set of system characteristics, such as process creation costs, and context switching overhead [21]. Saavedra and Smith use *microbenchmarks* to experimentally determine aspects of the system [25]. Gustafson and Snell [15] develop a scalable benchmark, HINT, that can accurately predict a machine’s performance via its memory reference capacity.

8 Conclusion

In an ideal world, static analysis would not only suffice, but would not limit the universe of approachable input codes. Unfortunately, we have experienced situations which break with this ideal, on one or both fronts: either static analysis fails to provide enough information to make good transformation decisions, or

the static analyses themselves preclude the very codes we desire to optimize. This paper has presented a modeling methodology which tackles these two problems.

References

1. D. Bailey and et al. NAS parallel benchmarks. <http://science.nas.nasa.gov/Software/NPB>.
2. D. A. Berson, R. Gupta, and M. L. Soffa. URSA: A unified resource allocator for registers and functional units in vliw architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL, Jan. 1993.
3. J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, 1997.
4. E. A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, 1994.
5. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. In *Symposium on Partial Evaluation*, 1998.
6. P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Programming Language Design and Implementation*, June 1997.
7. D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Principles of Programming Languages*, Saint Petersburg, FL, Jan. 1996.
8. A. P. Ershov. On the partial computation principle. *Inf. Process. Lett.*, 1977.
9. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Workshop on Languages and Compilers for Parallel Computing*, 1991.
10. B. Fraguera, R. Doallo, and E. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Parallel Architectures and Compilation Techniques*, Oct. 1999.
11. M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, Laboratory for Computer Science, Sept. 1997.
12. K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Supercomputing*, Nov. 1999.
13. S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behavior*. PhD thesis, Princeton, Sept. 1999.
14. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for c. Technical Report UW-CSE-97-03-03, University of Washington, Department of Computer Science and Engineering, June 1998.
15. J. L. Gustafson and Q. O. Snell. HINT—a new way to measure computer performance. In *HICSS-28*, Wailela, Maui, Hawaii, Jan. 1995.
16. W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *Proceedings of IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, Apr. 1995.
17. R. E. Ladner, J. D. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Symposium on Discrete Algorithms*, Jan. 1999.

18. M. Leone and P. Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software*, pages 8–17, Tuscon, AZ, 1996.
19. O. M. Lubeck, Y. Luo, H. J. Wasserman, and F. Bassetti. Development and validation of a hierarchical memory model incorporating cpu- and memory-operation overlap. Technical Report LA-UR-97-3462, Los Alamos National Laboratory, 1998.
20. J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, Dec. 1995.
21. L. McVoy and C. Staelin. Imbench: Portable tools for performance analysis. In *Usenix Proceedings*, Jan. 1995.
22. N. Mitchell. *Guiding Program Transformations with Modal Performance Models*. PhD thesis, University of California, San Diego, Aug. 2000.
23. N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. In *International Journal on Parallel Programming*, June 1998.
24. R. H. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Parallel Architectures and Compilation Techniques*, Boston, MA, Oct. 1996.
25. R. H. Saavedra and A. J. Smith. Measuring cache and tlb performance and their effect on benchmark run times. *IEEE Trans. Comput.*, 44(10):1223–1235, Oct. 1995.
26. V. Sarkar. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM J. Res. Dev.*, 41(3), May 1997.
27. V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations (Technical Summary). In *Programming Language Design and Implementation*, 1992.
28. O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing '93*, pages 410–419, Portland, Oregon, Nov. 1993.
29. M. J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing*, Toronto, CA, Aug. 2000.
30. T. P. Way and L. L. Pollock. Towards identifying and monitoring optimization impacts. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 1997.
31. R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing*, Nov. 1998.
32. D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Principles and Practice of Parallel Programming*, pages 137–146, Seattle, WA, Mar. 1990.