Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior

Jingling Xue, *Senior Member, IEEE*

and

Xavier Vera, *Student Member, IEEE*

To appear in IEEE Transactions on Computers.

• A preliminary version of this work was presented at HPCA-8 in February 2002 [54]. That earlier work was carried out during a period of the second author's PhD studies at UNSW. Please contact the first author in all future correspondence.

• J. Xue is with the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia. E-mail: jxue@cse.unsw.edu.au.

• X. Vera is with Institutionen för Datateknik, Mälardalens Högskola, Västerås, Sweden. E-mail: xavier.vera@mdh.se.

Abstract

Data caches are a key hardware means to bridge the gap between processor and memory speeds, but only for programs that exhibit sufficient data locality in their memory accesses. Thus, a method for evaluating cache performance is required to both determine quantitatively cache misses and to guide data cache optimizations. Existing analytical models for data cache optimizations target mainly isolated perfect loop nests. We present an analytical model that is capable of statically analyzing not only loop nest fragments but also complete numerical programs with regular and compile-time predictable memory accesses. Central to the whole-program approach are abstract call inlining, memory access vectors and parametric reuse analysis, which allow the reuse and interferences both within and across loop nests to be quantified precisely in a unified framework. Based on the framework, the cache misses of a program are specified using mathematical formulas and the miss ratio is predicted from these formulas based on statistical sampling techniques. Our experimental results using kernels and whole programs indicate accurate cache miss estimates in substantially shorter amount of time (typically several orders of magnitude faster) than simulation.

Index Terms

Modeling Techniques, Analytical Modeling, Cache Memories, Data Locality, Performance Evaluation

1 Introduction

Memory speeds in today's computers have fundamentally lagged behind processor speeds [26]. The increasing performance mismatch has required increasingly more levels of cache memories (e.g., three levels in the Intel IA-64 processors) to prevent performance degradation, in which each level trades off higher capacity for faster access times. There can be a wide performance gap between programs that are designed to optimize cache performance and those that are not. As a consequence, a program's cache behavior is becoming the major performance-determining factor. While optimization technology for improving instruction cache performance is relatively mature [23], [37], [50], data cache optimizations are still at an early stage. We believe the lack of a general-purpose analytical model for evaluating the data cache behavior of a program is the primary reason for this imbalance and the major obstacle to a systematic exploration of data cache optimizations.

Data cache behavior is very difficult to analyze both accurately and efficiently. Due to the non-linear mapping of memory accesses to cache sets, data cache behavior is inherently unstable and can fluctuate widely due to slight variations in problem size parameters and base addresses [32]. Thus, a realistic model must strike a balance between accuracy and efficiency in order to be viable in guiding a range of optimizations in a huge search space.

Trace-driven simulation [51] and hardware measurement [2] can obtain accurately a broad range of performance metrics for a program. But their precision and flexibility come at a price. The former consumes a large amount of simulation time and can demand gigabytes of space for storing address traces. The latter — while not quite as time-consuming — is obviously restricted to measurements of existing caches. In addition, both approaches inherently offer little insight into the causes behind cache misses.

To overcome or avoid these limitations, compiler writers have explored the use of heuristics-based models to optimize the cache performance of numerical programs, which typically consist of loop nests operating on arrays. A number of models have been used to guide optimizing compilers in their choice of loop and data transformations such as loop tiling or blocking [10], [31], [32], [45], [57], [59], loop permutation [38], loop fusion [13] and padding [4], [28], [29], [44], [48], [52]. While promising significant performance improvements for some programs, these models are mostly qualitative (in estimating cache misses) and often restricted to a particular optimization technique or individual nests.

A recent study [39] reports that most misses in numerical codes are inter-nest even though most reuse is intra-nest and suggests that "optimizations cannot simply focus on nest optimizations but need to consider more than one nest." When inter-nest optimizations are included, the resulting search space is even larger. In order to reason about the interactions between optimizations and apply them in concert, we need detailed knowledge about the frequency and causes of cache misses in the program. To that end, a general-purpose model is required to give quantitative measurement of cache misses for parts or the entirety of the program. Such a model is expected to represent a good compromise between simulation, which is expensive, and heuristics-based modeling, which can be imprecise.

In the past decade, several general-purpose analytical models targeting regular numerical codes have emerged.[1] These include the footprint-oriented models [25], [47], the Cache Miss Equations (CMEs) [22], the probabilistic model [20] and the Presburger-formulas-based model [7]. All except [7] are limited mainly to analyzing individual perfect nests with

[1]The *analytical* means that the underlying model relies on some form of mathematical analysis for predicting cache misses although some parts of the model can be algorithmic in nature.

straight-line assignments and *neither* handles the call statements in the program. All these models have been applied to small kernels containing only a few array references.

This paper presents an analytical model for statically predicting the cache behavior of numerical programs (or its selected regions) with regular and compile-time predictable memory accesses. The proposed model is applicable to $\mathbb{K}$-way set-associative data caches. Central to the whole-program approach are abstract call inlining, memory access vectors and parametric reuse analysis, which allow the reuse and interferences both within and across nests to be quantified precisely in a unified compile-time framework. The contributions and limitations of this work are summarized below.

• **Abstract Inlining.** In this preprocessing step, we abstractly inline all the calls in a program to obtain a sequence of loop nests free of calls. The inlined program contains exactly the same memory accesses executed in exactly the same order as in the original program. The inlined program is statically analyzable if the original program is. As an enabling technique, the technique may also be useful to other cache modeling techniques.

• **Memory Access Vectors.** We introduce the memory access vectors as a powerful abstraction for the memory accesses of array references across the entire program. A memory access vector encompasses three pieces of information: the reference for which the memory access is executed, the loop nest in which the reference is contained and the particular iteration at which the reference is accessed. The memory access vectors determine statically the order in which a program's memory accesses are executed.

• **Parametric Reuse Analysis.** We present an integer programming formulation that computes exactly, for a given memory access to a memory line, the most recent previous access (MRPA) also to the same line. We derive from this formulation a practical algorithm for uniformly generated references, which may be contained in distinct nests possibly guarded by arbitrary affine IF conditionals. This algorithm is efficient since it requires integer programming only rarely and exact in commonly occurring cases (Theorem 3). Our reuse analysis is parametric since the MRPA of an access is expressed parametrically as a function of its memory access vector.

• **Whole-Program Modeling.** Based on our parametric reuse analysis, we specify the cache misses of a program using mathematical formulas and predict its miss ratio efficiently from these formulas based on statistical sampling techniques [53]. Our model is applicable to numerical codes with regular and compile-time predictable accesses consisting of subroutines, calls, IF statements and arbitrarily (perfectly or imperfectly) nested loops. In order to obtain accurate predictions of miss ratios statically, these programs must be free of data-dependent constructs such as variable loop bounds, data-dependent IF conditionals, indirection arrays and recursive calls.

• **Prototyping Implementation.** We have implemented our techniques in a prototyping system, which consists of components inlining calls (abstractly), obtaining memory access vectors, performing reuse analysis, sampling memory accesses, forming the mathematical formulas for cache misses and solving them for obtaining miss ratios.

• **Validation and Experimental Results.** We have validated the accuracy of our model against simulation using loop nest fragments and complete programs from SPECfp95, Perfect Suite, Livermore kernels, Linpack and Lapack. The analysis time is generally several orders of magnitude faster than simulation. The memory requirements are very reasonable in lieu of the fast analysis time. The largest program we have analyzed, Applu from SPECfp95, has 3868 lines of code, 16 subroutines and 2565 references. In the case of an 8KB (direct-mapped, 2-way, 4-way and 8-way, resp.) data cache with a 64B cache line size, our model obtains the miss ratios with absolute errors (0.99%, 0.97%, 0.97% and 0.92%, resp.) in only 130 seconds using about 60.3MB memory while the cache simulation runs for about 5 hours on a 933MHz Pentium III PC. In comparison with the existing analytical models (reviewed in Section 2), our model is the only one with a demonstrated capability for analyzing programs of this scale efficiently with a good degree of accuracy.

This work has enlarged the scope of programs that can be analyzed statically from individual nests to whole programs. Admittedly, the proposed model is still limited to data-independent constructs. But it should be recognized that developing a general-purpose quantitative model for analyzing data-dependent constructs is an important next step. There are relatively few published results in this challenging endeavor, although some qualitative models for optimizing irregular codes can be found in [19], [24], [40], [49]. Unlike our (static) model, such a *dynamic* model will usually rely on some form of address traces for behavior analysis. We believe both types of models have their respective roles to play, just like how static and dynamic program analysis co-exist in staged or adaptive compilation.

The rest of this paper is organized as follows. Section 2 provides a detailed review of the related work. Section 3 describes our program model, the cache architecture used and our prototyping implementation for exercising and validating our analytical model. Section 4 introduces the three components — abstract call inlining, memory access vectors and parametric reuse analysis — central to the whole-program analytical modeling. Section 5 presents the mathematical formulas for specifying the cache misses of a program and discusses an algorithm for obtaining cache misses from these formulas. Section 6 presents experimental results. Section 7 concludes and discusses future work.

2 Related Work

Some earlier attempts on estimating cache misses at compile time can be found in [18], [21], [41], [57]. Below we review in detail the five *general-purpose analytical* cache models developed primarily for guiding data cache optimizations [5], [7], [20], [22], [25], [47], [53]. These models are all restricted to data-independent language constructs, and consequently, can obtain predictions of miss ratios at compile time without relying on address traces.

An analytical model consists of three components: *reuse analysis*, *cache miss specification* and *cache miss generation*. In some cases, some or all of the three are combined. Reuse analysis applies a reuse metric to obtain quantitative measurement of data reuse in the program. Based on this analysis, some mathematical formulas for specifying cache misses are set up. In the case of numerical programs, these formulas typically describe the relationships among loop variables, array sizes, base addresses and cache parameters. Finally, the cache miss information is generated from the specification by some means.

Temam *et al* [47] estimate the cache misses of individual perfect nests with rectangular iteration spaces for direct-mapped caches. They consider a subset of uniformly generated references so that all temporal reuse vectors are essentially basis vectors. To find the misses for a particular iteration of a reuse-carrying loop (corresponding to a basis reuse vector), they compute the footprint (i.e., the set of cache lines) accessed by each reference and solve their formulas expressed in terms of these footprints. Despite that the footprints are generally approximated, they obtain good estimates for several kernel examples. Recently, Harper *et al* [25] give an extension to $\mathbb{K}$-way set-associative caches but still for the same class of nests. They obtain good estimates for the four example kernels: MM as given in

Appendix B, a 2-D SOR nest, a 2-D Jacobi nest and a blocked matrix multiplication.[2]

Ghosh *et al* [22] introduce the well-known CMEs, a set of equalities and inequalities, to specify the cache misses of a single perfect nest with straight-line assignments for $\mathbb{K}$-way set-associative caches. Their reuse metric is Wolf and Lam's reuse vectors, which they obtain from uniformly generated references approximately by relying on Wolf and Lam's reuse framework and some ad hoc techniques. They show that the CMEs can help an optimizing compiler choose tile and pad sizes without requiring the CMEs to be solved explicitly. When there is a need to solve the CMEs for cache misses, Vera *et al* [53] discuss how to do so efficiently with a controlled degree of accuracy by using statistical sampling on a version of the CMEs greatly simplified by Bermudo *et al* [5] in the polyhedral model [16], [33]. Statistical techniques have also been applied to trace sampling to produce simulation results within an error margin with a pre-defined confidence [51].

Fraguela *et al* [20] rely on a probabilistic model to provide a fast estimation of cache misses for $\mathbb{K}$-way set-associative caches. They use the so-called area vectors as a reuse metric to represent probabilistically the amount of reuse along those directions and solve their recursive cache miss equations for cache misses. When analyzing imperfect nests, they exploit only the reuse between references contained in a common nest. These references differ by constants in their matching dimensions, forming a subset of uniformly generated references considered in the CMEs. They validate the accuracy of their model using three kernel examples. The two perfect nests can be analyzed by the CMEs and are not compared here. The third one is a 3-D blocked imperfect nest for computing AB^T (named MMT and given in Appendix B). Table 1 compares their model with ours.[3] Our *EstimateMisses* (given in Figure 7) produces better results in all cases. In both models, the two largest relative errors are due to the fact that the total number of misses is small in each case.

Chatterjee *et al* [7] present a cost model for exactly analyzing the cache behavior of loop nests for $\mathbb{K}$-way set-associative caches. They use Presburger formulas (as a reuse metric) to specify a program's cache misses, the Omega Calculator [42] to simplify the formulas,

[2]It is impractical to compare with this work. They present their average relative errors in Tables 1 and 2 of their paper without providing details on the samples taken. We have decided just to include our results for their MM in Section 6.

[3]The analysis times are not compared since the required data are not given in their paper [20]. In our model, each case takes less than 1 second. To compare with their results, we have used relative rather than absolute errors.

TABLE 1

Comparison with Fraguela et al's Model Using MMT in Appendix B for Various Cache Configurations $(\mathbb{C}, \mathbb{L}, \mathbb{K})$ — 4-Way Caches Were Not Considered by Them. In Each Case, Δ_P Denotes Their Relative Error between the Predicted and Simulated Miss Ratios and Δ_E for Our *EstimateMisses* given in Figure 7 with $c = 95\%$ and $w = 0.05$ as the Input Confidence and Interval, Respectively.

N	BJ	BK	$\mathbb{C}$ (KB)	$\mathbb{L}$ (B)	$\mathbb{K}$-Way	Δ_P (%)	Δ_E (%)
200	100	100	16	8	2	6.23	0.10
200	100	100	256	16	2	2.73	0.50
200	200	100	32	8	1	6.88	0.06
200	200	100	128	8	2	2.86	0.05
200	200	100	128	32	2	44.25	16.00
200	50	200	16	4	1	4.62	0.05
200	100	200	32	8	2	12.51	0.10
200	100	200	64	16	1	3.31	0.40
400	100	100	16	8	2	4.48	0.03
400	100	100	256	16	2	4.26	0.50
400	200	100	32	8	1	2.65	0.40
400	200	100	128	8	2	5.82	0.05
400	200	100	128	32	2	44.68	16.00
400	50	200	16	4	1	2.02	0.05
400	100	200	32	8	2	5.55	0.06
400	100	200	64	16	1	7.12	0.30

PolyLib [56] to obtain an indiscriminating union of polytopes, and finally, Ehrhart polynomials to count the integer points (i.e., misses) in each polytope [9]. They can formulate Presburger formulas for a looping structure consisting of imperfect nests, IF statements, references with arbitrary affine accesses and non-linear data layouts. However, their current implementation as a SUIF [36] pass "does not yet have enough performance to be practical." Two examples are discussed: matrix multiplication — it takes 1 to 10 seconds on a 300MHz Sparc Ultra 60 to analyze the 21×21 instance; matrix-vector product — they have derived the Presburger formulas for $N = 100$ but did not solve them.

3 Background and Terminology

3.1 Program Model

Our analytical model applies to numerical programs consisting of subroutines, calls, arbitrarily (perfectly or imperfectly) nested loops, and assignments possibly guided by IF conditionals. The typical data structures in these programs are 1-D and 2-D arrays.

Definition 1 (References). *A* **reference** *is a static read or write in the program.*

Definition 2 (Memory Accesses). *A* **memory access** *is the execution of a reference at a particular iteration of the loop nest enclosing the reference.*

Definition 3 (Affine Expressions). *An expression in a program is* **affine** *if it has the form $c_1 I_1 + \cdots + c_n I_n + b$, where $I_1, \ldots, I_n$ are the loop variables of the n enclosing loops (if any) and $c_1, \ldots, c_n, b$ are compile-time or runtime constants.*

We rely on the optimizing compiler to identify compile-time and runtime constants. In numerical codes, some variables are initialized once and never subsequently modified. Detecting these runtime constants allows more affine expressions to be recognized.

In this paper, all programs are in FORTRAN 77. All example codes are expressed in a FORTRAN-like syntax. Thus, all arrays are assumed to be in column major, but the techniques apply to any *linear* data layout.[4] All the load/store references are assumed to be arrays. Scalars are either register-allocated or considered as a special case of 1-D arrays.

The following restrictions define the scope of programs considered in this work:

- The calls are non-recursive (which is the case in FORTRAN 77).
- The bounds of all loops are affine.[5]
- The IF conditionals guiding array references are affine.[6]
- The subscript expressions of array references are affine.

These constraints ensure that our analysis can be done in the polyhedral model [16], [33]. They are not too restrictive for static analytical modeling, which is generally limited to the statically predictable control flow and memory accesses (independently of the input data). Our program model excludes essentially only data-dependent constructs, which include, for

[4]An m-D array can be stored linearly in $m!$ different ways including row- and column-major as two special cases.

[5]As a consequence, the input parameters initialized in READ statements are treated as runtime constants.

[6]In our model, the IF conditionals involving mod/div/floor/ceiling can be handled. The basic idea is to make these expressions linear by introducing extra free variables [42]. For example, $I \bmod 4 = 1 \equiv I - 4q = 1 \wedge q \in \mathbb{Z}$. To avoid complicating matters in several places in our presentation, their discussions are omitted.

example, recursive calls, variable loop bounds, data-dependent IF conditionals and indirection arrays. Recursive calls are data-dependent since they are usually terminated by data-dependent IF conditionals. These constraints are assumed in the prior work reviewed in Section 2 except that some non-linear array layouts can be accommodated in [7].

3.2 Cache Model

We consider a uniprocessor with a two-level memory hierarchy consisting of a virtually-indexed set-associative data cache using LRU replacement followed by main memory. In the case of write misses, we assume a fetch-on-write policy so that reads and writes are not distinguished. Many systems utilize physically-indexed caches. An extension of this work considering the effects of page placement [30] is a future work. Note that this cache architecture is assumed in all the existing analytical work reviewed in Section 2.

In a $\mathbb{K}$-way set-associative cache, a cache set contains $\mathbb{K}$ distinct cache lines. Let $\mathbb{C}$ ($\mathbb{L}$) be the cache (line) size in bytes. The total number of cache sets is thus $\mathbb{C}/(\mathbb{L}\times\mathbb{K})$. Sometimes, a cache configuration is identified as a triple $(\mathbb{C},\mathbb{L},\mathbb{K})$, which is *direct-mapped* if $\mathbb{K}=1$ and *fully-associative* if $\mathbb{K}=\mathbb{C}/\mathbb{L}$ in the two extreme cases.

$\mathbb{L}$ is in bytes rather than array elements so that our formulas on reuse analysis and cache miss analysis work directly when the arrays involved have different element sizes.

A *memory line* refers to a cache-line-sized block in the memory while *a cache line* refers to the actual block in which a memory line is mapped.

Definition 4 (Reuse). *When an (array) reference R accesses a memory line that was accessed previously by an (array) reference R', it is called a* **reuse** *of that memory line by the reference R [22]. The reuse is* **temporal** *if the same data element is accessed in both cases and* **spatial** *otherwise and the reuse is* **self** *if $R = R'$ and* **group** *otherwise*[7] *[57].*

Following the CMEs [22], cold misses are used in the normal manner but capacity and conflict misses are combined and called *replacement misses.*

3.3 Compilation Model

Our model analyzes statically possible cache conflicts in a program. Therefore it needs to know the addresses of all memory accesses at compile time. In addition to the program

[7]This classification can be generalized in a meaningful way if the elements accessed have different element sizes.

restrictions stated in Section 3.1, the following ones must also be respected:

- The base addresses of all arrays are known statically. If A is an array variable, the notation $@A$ stands for its base address in the rest of this paper.
- The sizes of all arrays in all but their last dimensions are known statically.

Both together ensure that all affine array references have statically known addresses. In fact, all arrays must be statically allocated in order to be analyzable. Due to the call-by-reference semantics, the dummy arrays that satisfy these constraints are dealt with by the abstract inlining technique in Section 4.1. The arrays that appear in COMMON or EQUIVALENCE statements are not special except that the reuse between the references to these aliased arrays can be more expensive to analyze exactly if they have different dimensions, shapes or extents (which are bad FORTRAN programming styles, in general).

Figure 1 depicts the structure of our prototyping system for *exercising* our analytical model and *validating* its accuracy against simulation. Our model consists of the five components represented by the doubly-framed boxes. The program being analyzed is first translated by a Polaris parser into Polaris IR [14], which is subsequently translated by Ictineo [3] into a so-called Ictineo load/store IR. The lowering of the Polaris IR serves two purposes. First, some standard compiler optimizations such as constant propagation and induction variable elimination are performed so that the estimated miss ratio is a realistic representation of the miss ratio of the compiled code. Second, the load/store array references are identified and the scalars and temporaries mapped to virtual registers. Once both are done, the load/store IR is converted to an optimized Polaris IR in which all load/store array references are clearly indicated. This same code will be analyzed by our model to obtain the estimated miss ratio and also instrumented to obtain the simulated miss ratio.

It should be pointed out that our model is not limited to the Polaris and Ictineo IRs. The model can be applied to any IRs as long as the information required by the model is available. In addition, scalars and temporaries are not required to be register-allocated. As mentioned in Section 3.1, the memory-allocated scalars can be considered as a special-case of 1-D arrays. In Section 4.1, we will see that the stack-allocated temporaries can be mapped to the appropriate elements of the runtime stack due to the absence of recursion.

Most optimizing compilers keep loop variables in registers. We state clearly the following simplifying assumption made in all our experiments. We make comments wherever

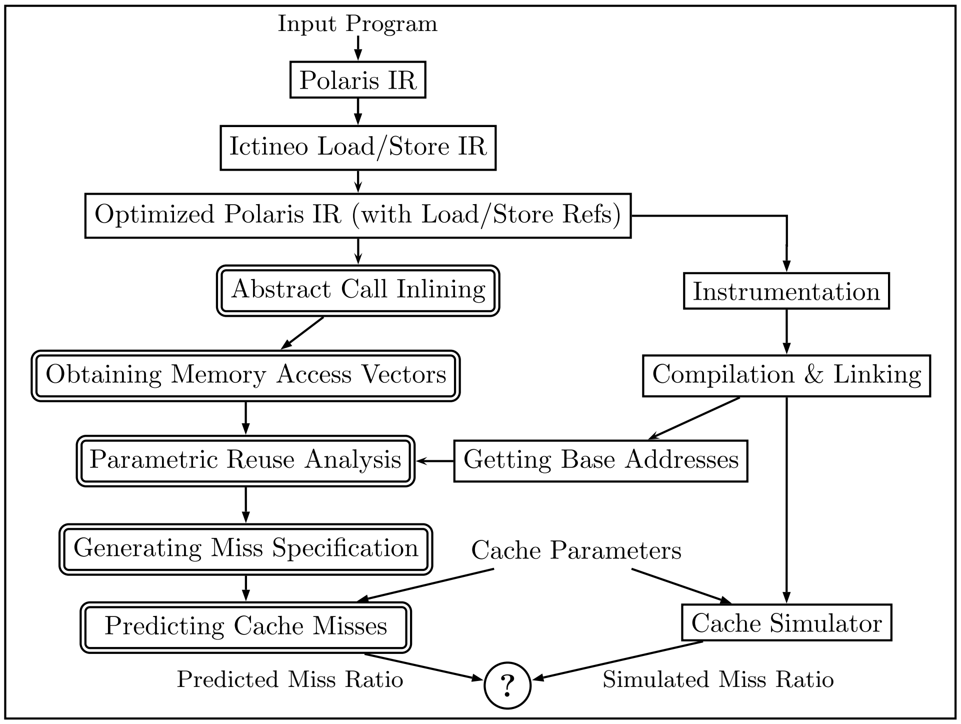


Fig. 1. A framework for exercising and validating our analytical model.

appropriate to show that our model works straightforwardly when this assumption is lifted.

Assumption 1: *All loop variables are assumed to be register-allocated.*

As a result, a loop variable will not induce any memory accesses in behavor analysis.

4 Whole-Program Analysis

Analytical modeling relies on the following fact or its variants to make static predictions.

Theorem 1: *In a $\mathbb{K}$-way set-associative cache with LRU replacement, these two statements are true. (a) A memory access m_a to memory line ℓ is a cold miss if ℓ is accessed for the first time. (b) Let m_b be the most recent previous access (MRPA) also to ℓ. Then m_a is a replacement miss if there are $\mathbb{K}$ or more distinct memory lines that are accessed between m_a and m_b that are also mapped to the same cache set as ℓ and a hit otherwise.*

To apply this theorem, we need to achieve the following two goals statically:

- Identify two consecutive accesses (the earlier being its MRPA) to a memory line.
- Identify all intervening accesses between such a pair of consecutive accesses.

Our compile-time framework for achieving these two goals, which consists of abstract call inlining, memory access vectors and parametric reuse analysis, is described below.

```
REAL*8 $\mathcal{A}(N_1, \ldots, N_{m-1}, *)$
    ...
CALL $foo(\ldots, \mathcal{A}(f_1, \ldots, f_m), \ldots)$
    ...
SUBROUTINE $foo(\ldots, \mathcal{F}, \ldots)$
REAL*8 $\mathcal{F}(M_1, \ldots, M_{p-1}, *)$
    ...
  $\mathcal{F}(g_1, \ldots, g_p)$
    ...
END
```

(a) A program segment

```
$Stack[BP]$ = RetAddr
$Stack[BP+4]$ = addr of 1st arg
$Stack[BP+8]$ = addr of 2st arg
      ...
$\ldots = Stack[BP+4]$
$\ldots = Stack[BP+8]$
      ...
  $\boxed{foo\text{'s code body goes here with the transformations below:}}$
    ■ dummy arrays:
    if $(p = 1 \ \vee \ (p = m \ \wedge \ (\forall\, 1 \leqslant k < m : N_k = M_k)))$
        • each reference $\mathcal{F}(g_1, \ldots, g_p)$ is replaced with:
          if $(p = 1)$
            $\mathcal{A}(f_1 + g_1 - 1, f_2, \ldots, f_m)$ //①
          else // $p = m \cdots$
            $\mathcal{A}(f_1 + g_1 - 1, \ldots, f_m + g_m - 1)$ //②
        • $\mathcal{F}$'s decl ignored
    else
        • each reference $\mathcal{F}(g_1, \ldots, g_p)$ is replaced with:
            $foo\_\mathcal{A}(\sum_{i=1}^{m}(\Pi_{k=1}^{i-1} N_k)(f_i - 1) + g_1, g_2, \ldots, g_p)$ //③
        • $foo\_\mathcal{A}$ has the same base address as $\mathcal{A}$
        • $foo\_\mathcal{A}$ has the same decl as $\mathcal{F}$
    ■ non-dummy arrays:
      • base addresses, decls, and refs to them are preserved
      • Each local array, $X$, is renamed uniquely as $foo\_X$
  RetAddr = $Stack[BP]$
```

(b) Inlined code

Fig. 2. Abstract inlining of a call statement.

4.1 Abstract Call Inlining

The objective is to obtain a program consisting of loop nests free of calls. This will allow the reuse and interferences across subroutines to be quantified precisely.

In FORTRAN 77, all arguments are passed by reference. To analyze a program containing calls, we perform an *abstract inlining* for all the calls recursively. We do not actually generate the inlined code. We need only to obtain the information required for analyzing the inlined code. Each subroutine is associated with an *abstract function* consisting of the information about the accesses to the runtime stack, its code body (i.e., its loop nests with references), and the base addresses and declarations for its dummies and other variables.

Figure 2 shows that the inlining of a call consists of essentially replacing the call with the information in the callee's abstract function. In the program fragment shown in Figure 2(a), the actual $\mathcal{A}(f_1, \ldots, f_m)$ and a generic reference $\mathcal{F}(g_1, \ldots, g_p)$ to the matching dummy $\mathcal{F}$ are affine. If $\mathcal{A}$ is the actual (without any subscripts), $\mathcal{A}$ is the same as $\mathcal{A}(1, \ldots, 1)$. Recall that memory-allocated scalars (including actuals and dummies) are considered 1-D arrays. The two issues addressed by the abstract inlining are discussed below.

• **Modeling of Accesses to the Runtime Stack.** The calling conventions are compiler-

```
REAL*8 $A(10,10)$
DO $I_1 = \ldots$
  DO $I_2 = \ldots$
    $A(I_1,I_2) = A(I_1,I_2-1)+1$
    CALL $f(A, A(I_1,I_2), A(I_1,I_2))$
    CALL $f(A(1,I_2), A(I_1-1,I_2), A)$
END
SUBROUTINE $f(B,C,D)$
REAL*8 $B(10), C(10,10), D(5,5,*)$
DO $I_3 = \ldots$
  DO $I_4 = \ldots$
    $\cdots = C(I_3,I_4-1)+C(I_3,I_4)$
    $D(I_3,I_4,2) = B(I_4)$
END
```

(a) Original program

```
REAL*8 $A(10,10), f\_A(5,5,*)$
// $f\_A$ has the same base addr as $A$: $@f\_A = @A$
DO $I_1 = \ldots$
  DO $I_2 = \ldots$
    $A(I_1,I_2) = A(I_1,I_2-1)+1$
    DO $I_3 = \ldots$
      DO $I_4 = \ldots$
        $\cdots = A(I_1+I_3-1, I_2+I_4-2) + A(I_1+I_3-1, I_2+I_4-1)$
        $f\_A(I_1+10*(I_2-1)+I_3-1, I_4, 2) = A(I_4,1)$
    DO $I_3 = \ldots$
      DO $I_4 = \ldots$
        $\cdots = A(I_1+I_3-2, I_2+I_4-2) + A(I_1+I_3-2, I_2+I_4-1)$
        $f\_A(I_3,I_4,2) = A(I_4,I_2)$
END
```

(b) Inlined version

Fig. 3. Transformations of the references to dummy arrays.

and architecture-dependent. What is shown in Figure 2(b) is one such a convention for a 32-bit machine, in which all actual arguments are passed via the runtime stack. *Stack* denotes the runtime stack modeled as a 1-D array. If BP is 0 initially, its value is known at compile time at every call site due to the absence of recursion. The base address of *Stack*, if unknown at compile time, has to be obtained at run time. Then *Stack* is treated just like an ordinary array. In Section 3.3, we mentioned that the stack-allocated temporaries may be introduced in a load/store IR. It is not difficult to see that they are mapped straightforwardly to their unique elements in the *Stack* array.

• **Transformation of Array References in the Callee.** As shown in Figure 2(b), the references to the non-dummy arrays remain unchanged. But the references to the dummy arrays are transformed so that the subscript expressions of the matching actuals are incorporated into the transformed references. All the three cases are illustrated in Figure 3. The inlined code may not compile. Hence, the term abstract inlining.

The inlined code is statically analyzable if the original program is. The inlined code contains the same memory accesses executed in the same order as in the original program. Both statements are true simply because the transformed references in Figure 2(b) are affine if both $\mathcal{A}(f_1,\ldots,f_m)$ and $\mathcal{F}(g_1,\ldots,g_p)$ are, and in addition to these transformations, the abstract inlining has not modified anything else in the original program. If Assumption 1 is lifted, the loop variables in $f_1,\ldots,f_m$ introduced into the transformed references are simply tagged so that they do not result in any superfluous memory accesses in analysis.

In our current implementation, we do not consider the cache effects of the system calls (to I/O subroutines and intrinsic functions). Only the memory accesses to their actual arguments are included. These calls can be inlined if their abstract functions are known.

While we inline in order to analyze cache misses exactly, existing techniques such as [1], [6], [46] inline primarily to optimize program performance.

4.2 Defining the Address Trace Statically

The inlined program has a flat structure consisting of multiple loop nests without any calls. The memory access vectors are defined to introduce a total order among all the memory accesses from both within a nest and across nests in the abstractly inlined program. They define precisely the address trace of the program at compile time.

4.2.1 Loop Nest Normalization

We normalize all loop nests to put the program into a suitable form for analysis. We apply loop sinking to move all statements into their respective innermost loops by adding IF conditionals wherever appropriate [57], [58]. We add trivial loops with equal lower and upper bounds, if necessary, so that all (deepest) loop nests are n-dimensional. Finally, all loop variables at depth k are normalized to I_k. From now on, whenever we speak of a loop nest, we mean an n-dimensional nest with all statements nested inside its innermost loop.

The normalized program contains the same accesses executed in the same order as in the original program. This is guaranteed by the semantics of loop sinking and Assumption 1. If this assumption is lifted, our normalization procedure works as usual except the loop variables in the extra IF conditionals and loops introduced during the normalization are simply tagged so that they do not result in any superfluous memory accesses in analysis.

4.2.2 Memory Access Vectors

The access of a reference in the program can be uniquely identified by (a) the loop nest in which the reference is contained, (b) the iteration of the nest at which the reference is accessed, and (c) the access order of the reference in the nest.

The loops in the program are identified hierarchically in the same way as how the sections in this paper are numbered except that, for example, we use a vector (x, y, z) to identify a loop while we would write $x.y.z$ to identify the section in the same lexical position.

PARAMETER (N=20)	**Access Vector**
REAL*8 B(N,N), C(N)	
DO I_1 = 1, N	
DO I_2 = 1, N	
IF (I_2.GE.2)	
$B(I_2-1,I_1)=\cdots$	$(1,i_1,1,i_2,1)$
IF (I_2.GE.2 .AND. I_1+I_2.GE.10)	
$\cdots=D(I_2-1)$	$(1,i_1,1,i_2,2)$
DO I_2 = 1, N	
$\cdots=B(I_2,I_1)$	$(1,i_1,2,i_2,1)$
DO I_1 =1, N	
DO I_2 = 1, N	
$D(I_2)=\cdots$	$(2,i_1,1,i_2,1)$

Fig. 4. Memory access vectors for an example.

Definition 5 (Loop Vectors). *Each n-dimensional loop nest is identified by the* **loop vector** *of its innermost loop, $(\ell_1,\dots,\ell_n)$, where ℓ_k means that the k-th loop of the nest is the ℓ_k-th (counted from 1) among all loops enclosed in the $(k-1)$-st loop.*

Definition 6 (Iteration Vectors). *The execution of an n-dimensional nest when $I_1 = i_1,\dots,I_n=i_n$, known as an iteration, is identified by the* **iteration vector** $\vec{\imath}=(i_1,\dots,i_n)$.

Recall from Figure 1 the access order of references is specific to a compiled code.

Definition 7 (Access Order of References). *Let $S_{\vec{L}}$ be the set of all references in a loop nest $\vec{L}$. We define $\alpha : S_{\vec{L}} \to \mathbb{Z}^+$ such that $\alpha(R)=k$, meaning that R is the k-th accessed reference (with any guarding IF conditional ignored) in a common iteration of $\vec{L}$.*

The memory accesses introduced in Definition 2 are specified by integer vectors.

Definition 8 (Access Vectors). *Let R be a reference in the nest $\vec{L}=(\ell_1,\dots,\ell_n)$. The access of R at the iteration $\vec{\imath}=(i_1,\dots,i_n)$ of the nest is identified uniquely by the* **(memory) access vector** $(\ell_1,i_1,\dots,\ell_n,i_n,\alpha_{\vec{L}}(R)) \in \mathbb{Z}^{2n+1}$.

Figure 4 lists the access vectors of the four references for an example program.

Definition 9 (Reference Iteration Spaces). *The* **reference iteration space (RIS)** *of a reference R, denoted $RIS_R \subset \mathbb{Z}^n$, is the n-dimensional polytope defined by the bounds of its n enclosing loops and its guiding IF conditional, if any.*

While the RISs for most references in numerical codes are convex, some are not. This can happen, for example, when a reference is guided by I_k`.LE.5.OR.`I_k`.GE.100`. Due to the program restrictions stated in Section 3.1, all loop bounds and IF conditionals are affine. Thus, each RIS is always expressible as a union of convex polytopes.

The set of all accesses of a reference R contained in the nest $(\ell_1, \ldots, \ell_n)$ is defined by:

$$\mathcal{M}_R \;=\; \{(\ell_1, i_1, \ldots, \ell_n, i_n, \alpha_{(\ell_1,\ldots,\ell_n)}(R)) \mid (i_1, \ldots, i_n) \in \mathit{RIS}_R\} \tag{1}$$

If *RefSet* is the set of all references, then the set of all accesses in the program is given by:

$$\mathcal{M} \;=\; \bigcup_{R \,\in\, \mathit{RefSet}} \mathcal{M}_R \tag{2}$$

Example 1: *Consider the two C references in Figure 4. By Definition 9,* $\mathit{RIS}_{D(I_2-1)} = \{(i_1, i_2) \mid 1 \leqslant i_1 \leqslant 20, 2 \leqslant i_2 \leqslant 20, i_1 + i_2 \geqslant 10\}$ *and* $\mathit{RIS}_{D(I_2)} = \{(i_1, i_2) \mid 1 \leqslant i_1, i_2 \leqslant 20\}$. *By (1), we obtain* $\mathcal{M}_{D(I_2-1)} = \{(1, i_1, 1, i_2, 2) \mid 1 \leqslant i_1 \leqslant 20, 2 \leqslant i_2 \leqslant 20, i_1 + i_2 \geqslant 10\}$ *and* $\mathcal{M}_{D(I_2)} = \{(2, i_1, 1, i_2, 1) \mid 1 \leqslant i_1, i_2 \leqslant 20\}$. *The connection between* RIS_R *and* $\mathcal{M}_R$ *is clear.*

Theorem 2: *If* $\vec{a}, \vec{b} \in \mathcal{M}$, *then the access* $\vec{b}$ *occurs before the access* $\vec{a}$ *iff* $\vec{b} \prec \vec{a}$.

Proof: Follows from Definitions 5 — 9, (1) and (2). ■

Thus, $\mathcal{M}$ possesses the (strict) total order, $\prec$, which is known as the lexicographic or dictionary order. This order specifies statically the temporal order in which all accesses in the program are executed. In other words, the accesses in $\mathcal{M}$ ordered by $\prec$ give rise to precisely the address trace of the program at compile time.

4.2.3 Notations

Let $\vec{a}$ be an access of a reference R. We write $\mathit{ma}_R(\vec{a})$ to represent the memory address in *bytes* of the element accessed at the access $\vec{a}$. Let $\mathit{ml}_R(\vec{a})$ and $\mathit{cs}_R(\vec{a})$ be the memory line and cache set to which $\mathit{ma}_R(\vec{a})$ is mapped, respectively. In each case, the redundant subscript R serves to highlight the reference of which $\vec{a}$ is an access. It is understood that $\mathit{ma}_R(\vec{a})$ can be calculated in the standard manner. The other two terms are given by:

$$\mathit{ml}_R(\vec{a}) \;=\; \lfloor \mathit{ma}_R(\vec{a})/\mathbb{L} \rfloor \tag{3}$$

$$\mathit{cs}_R(\vec{a}) \;=\; \mathit{ml}_R(\vec{a}) \bmod (\mathbb{C}/(\mathbb{L} \times \mathbb{K})) \tag{4}$$

Example 2: *For the first B reference in Figure 4, we have* $\mathit{ma}_{B(I_2-1,I_1)}(1, i_1, 1, i_2, 1) = @B + 8(i_2 - 2 + N(i_1 - 1))$, $\mathit{ml}_{B(I_2-1,I_1)}(1, i_1, 1, i_2, 1) = \lfloor (@B + 8(i_2 - 2 + N(i_1 - 1)))/\mathbb{L} \rfloor$ *and* $\mathit{cs}_{B(I_2-1,I_1)}(1, i_1, 1, i_2, 1) = \lfloor (@B + 8(i_2 - 2 + N(i_1 - 1)))/\mathbb{L} \rfloor \bmod (\mathbb{C}/(\mathbb{L} \times \mathbb{K}))$.

If $\vec{a}$ is an access vector, $\vec{a}_I$ identifies the iteration at which the access is executed.

4.3 Parametric Reuse Analysis

This section computes the MRPA (most recent previous access) as required in Theorem 1. We give below a precise and insightful definition for the problem addressed in reuse analysis.

We postulate the existence of an undefined access vector $\perp \notin \mathcal{M}$ that is always executed before any other accesses: $\forall\, \vec{a} \in \mathcal{M} : \perp \prec \vec{a}$. Let S be a set of integer vectors. The notation $\max_{\prec} S$ denotes the *lexicographic maximum* of S. By convention, $\max_{\prec} \emptyset = \perp$.

Let R and R' be two arbitrary references, which are not necessarily different. We define:

$$\mathcal{P}_{R,R'}(\vec{a}) = \{\vec{b} \in \mathcal{M}_{R'} \mid \vec{b} \prec \vec{a}, ml_R(\vec{a}) = ml_{R'}(\vec{b})\} \tag{5}$$

which consists of all accesses of R' preceding $\vec{a}$ and mapped to the same line as $\vec{a}$. Let

$$ipred_{R,R'} : \mathcal{M}_R \rightarrow \mathcal{M}_{R'} \cup \{\perp\}, \quad ipred_{R,R'}(\vec{a}) = \max_{\prec} \mathcal{P}_{R,R'}(\vec{a}) \tag{6}$$

Then $ipred_{R,R'}(\vec{a})$ is the most recent previous access, i.e., the one that *immediately precedes* $\vec{a}$ in $\mathcal{P}_{R,R'}(\vec{a})$. Note that $ipred_{R,R'}(\vec{a}) = \perp$ holds as desired when $\mathcal{P}_{R,R'}(\vec{a}) = \emptyset$.

By composing the $ipred_{R,R'}$ functions for a fixed R but varying R', we get:

$$ipred_R : \mathcal{M}_R \rightarrow \mathcal{M} \cup \{\perp\}, \quad ipred_R(\vec{a}) = \max_{\prec}\{ipred_{R,R'}(\vec{a}) \mid R' \in RefSet\} \tag{7}$$

The reuse analysis is concerned with finding a good approximation of the function $ipred_R$ *for every reference in the program.* In fact, $\vec{a}$ and $ipred_R(\vec{a})$ play exactly the roles that the access m_a and its MRPA m_b played in Theorem 1, respectively.

We write $\mathsf{MRPA}_{R,R'}$ for our approximation of $ipred_{R,R'}$. As a result, MRPA_R is our approximation of $ipred_R$. $\mathsf{MRPA}_{R,R'}$ (MRPA_R) is *exact* if $\mathsf{MRPA}_{R,R'} = ipred_{R,R'}$ ($\mathsf{MRPA}_R = ipred_R$). In Section 4.3.1, we give an integer programming formulation for finding $ipred_{R,R'}$ exactly. In Section 4.3.2, we derive from this formulation a practical algorithm for finding $\mathsf{MRPA}_{R,R'}$ mostly analytically by considering uniformly generated references. In Section 4.3.3, we discuss various trade-offs that can be made between efficiency and accuracy in our reuse framework. In Section 4.3.4, we discuss the equivalence of two reuse metrics, MRPA and *shortest* reuse vectors, and highlight the importance of this work. Appendix A discusses strategies for minimizing the time on evaluating the MRPA functions in behavior analysis.

4.3.1 Constructing ipred$_{R,R'}$

From our definition of the function $ipred_{R,R'}$, we recognize that we can build the function exactly using parametric integer programming (PIP) [15], [35] by solving:

$$ipred_{R,R'}(\vec{a}) \quad = \quad \max{}_{\prec} \left\{ \vec{b} \;\middle|\; \begin{array}{l} \vec{a} \triangleq (\ell_1, i_1, \ldots, \ell_n, i_n, \alpha_{(\ell_1,\ldots,\ell_n)}(R)) \in \mathcal{M}_R \\ \vec{b} \triangleq (\ell'_1, j_1, \ldots, \ell'_n, j_n, \alpha_{(\ell'_1,\ldots,\ell'_n)}(R')) \in \mathcal{M}_{R'} \\ \vec{b} \prec \vec{a} \\ ml_R(\vec{a}) = ml_{R'}(\vec{b}) \end{array} \right\} \tag{8}$$

where the n loop variables of $\vec{b}_I = (j_1, \ldots, j_n)$ are constraint variables, the n loop indices of $\vec{a}_I = (i_1, \ldots, i_n)$ are structural parameters (i.e., symbolic constants) and all the rest (including $\mathbb{L}$, base addresses and array sizes appearing in $ml_R(\vec{a})$ and $ml_{R'}(\vec{b})$) are integers.

Our model does not require (8) to be solved directly in that form. We explain briefly how to solve it by PIP and then give an example. The domain of (8) is not convex. However, we can always transform the problem into a finite number of subproblems whose domains are convex polytopes and obtain $ipred_{R,R'}(\vec{a})$ from the lexicographic maximums of these polytopes. The transformations are as follows. Every RIS is always expressible as a union of convex polytopes (Section 4.2). The non-linear constraint $ml_R(\vec{a}) = ml_{R'}(\vec{b})$ can be linearized as illustrated in Example 3. Let R and R' be enclosed in d common loops. Then $(\ell_1, \ldots, \ell_d) = (\ell'_1, \ldots, \ell'_d)$. Clearly, $\vec{b} \prec \vec{a} \equiv (\vee_{k=1}^{d} \vec{b} \prec_{2k} \vec{a}) \vee (\vec{b} \prec_{2d+1} \vec{a})$, where

$$\vec{b} \prec_k \vec{a} \quad \equiv \quad b_1 = a_1 \wedge \cdots \wedge b_{k-1} = a_{k-1} \wedge b_k < a_k \tag{9}$$

Example 3: *To construct* ipred$_{D(I_2),D(I_2-1)}$ *for the program given in Figure 4, we assume that* $\mathbb{L} = 32$ *and* $@D = 3200$. *By letting* $\vec{a} = (2, i_1, 1, i_2, 1)$ *and* $\vec{b} = (1, j_1, 1, j_2, 2)$, *we find that* $\vec{b} \prec \vec{a}$ *is always true.* $\mathcal{M}_{D(I_2)}$ *and* $\mathcal{M}_{D(I_2-1)}$ *found in Example 1 are convex. In addition,* $ml_{D(I_2)}(2, i_1, 1, i_2, 1) = \lfloor (3200 + 8(i_2 - 1))/32 \rfloor$ *and* $ml_{D(I_2-1)}(1, j_1, 1, j_2, 2) = \lfloor (3200 + 8(j_2 - 2))/32 \rfloor$. *By linearizing* $ml_{D(I_2)}(2, i_1, 1, i_2, 1) = ml_{D(I_2-1)}(1, j_1, 1, j_2, 2)$ *with the introduction of a new constraint variable, say,* q, *we transform (8) to get:*

$$(ipred_{D(I_2),D(I_2-1)}(\begin{bmatrix} 2 \\ i_1 \\ 1 \\ i_2 \\ 1 \end{bmatrix}), q) \quad = \quad \max{}_{\prec} \left\{ \begin{bmatrix} 1 \\ j_1 \\ 1 \\ j_2 \\ 2 \\ q \end{bmatrix} \;\middle|\; \begin{array}{l} 1 \leqslant i_1, i_2, j_1 \leqslant 20 \\ 2 \leqslant j_2 \leqslant 20 \\ j_1 + j_2 \geqslant 10 \\ 0 \leqslant i_2 - 4q + 99 \leqslant 3 \\ 0 \leqslant j_2 - 4q + 98 \leqslant 3 \\ q \geqslant 0 \end{array} \right\} \tag{10}$$

Solving the system parametrically, we obtain $\text{ipred}_{D(I_2),D(I_2-1)}(2, i_1, 1, i_1, 1) = (1, 20, 1, 4((i_2 + 3) \div 4) + 1, 2)$ *if* $i_2 \leqslant 16$ *and* $(1, 20, 1, 20, 2)$ *otherwise.*

4.3.2 Constructing $\mathsf{MRPA}_{R,R'}$

It is unnecessary and can be expensive to solve (8) as a form of PIP for a large number of reference pairs. In numerical codes, the patterns of accesses are regular [39]. We can capture most of this regularity by considering only the accesses from uniformly generated references [21], [57], [60] that are generalized from single to multiple nests.

Definition 10 (Uniformly Generated References/Sets). *Let A be an m-D array. Two references $A(H_1\vec{I} + \vec{c}_1)$ and $A(H_2\vec{I} + \vec{c}_2)$ (inside the same or distinct n-dimensional nests) are* **uniformly generated** *if $H_1 = H_2$, where $H_1, H_2 \in \mathbb{Z}^{m \times n}$ and $\vec{c}_1, \vec{c}_2 \in \mathbb{Z}^m$. The* **uniformly generated set** *for a reference R, denoted $UGR(R)$, consists of all references (including R itself) in the program that are uniformly generated.*

In the program given in Figure 4, two uniformly generated sets are $\{B(I_2-1, I_1), B(I_2, I_1)\}$ and $\{D(I_2 - 1), D(I_2)\}$. In the two kernels Hydro_K and MGRID_K given in Appendix B, there are a few uniformly generated sets consisting of references from different nests.

Definition 11 (Uncoupled References). *A reference $A(H\vec{I} + \vec{c})$ or H is* **uncoupled** *if each row of H has at most one nonzero component, where $H \in \mathbb{Z}^{m \times n}$ and $\vec{c} \in \mathbb{Z}^m$.*

We have developed a practical algorithm, named *FindMRPA* in Figure 5, for computing $\mathsf{MRPA}_{R,R'}$ when R and R' are uniformly generated. This algorithm is exact in commonly occurring cases (Theorem 3). It addresses the three cases classified in lines 15, 23 and 39 separately. Of all uniformly generated reference pairs in SPECfp95 and Perfect Benchmarks, the frequencies of these cases are 75.41%, 22.53% and 2.06%, respectively. *FindMRPA* solves the first two cases analytically and requires integer programming in the last case not only rarely but also on a simplified version of (8).

Based on the notations in lines 2 – 9 in *FindMRPA*, we explain its basic idea, examine its three cases, and finally, illustrate each by an example. The simple fact we rely on is:

$$ml_R(\vec{a}) = ml_{R'}(\vec{b}) \;\Rightarrow\; |ma_R(\vec{a}) - ma_{R'}(\vec{b})| < \mathbb{L} \div \mathbb{E} \tag{11}$$

where $\mathbb{L} \div \mathbb{E}$ is the number of elements in a memory line. But the converse is obviously not true. Next, we divide all the accesses of R' preceding $\vec{a}$ into disjoint sets based on their

```
Algorithm FindMRPA
INPUT:   $R \triangleq A(H\vec{I}+\vec{c})$: a reference nested in $\vec{L}=(\ell_1,\ldots,\ell_n)$
         $R' \triangleq A(H\vec{I}+\vec{p})$: a reference nested in $\vec{L}'=(\ell'_1,\ldots,\ell'_n)$
         $\vec{h}_k$: the $k$-th row of $H \in \mathbb{Z}^{m\times n}$
         $c_k$: the $k$-th component of $\vec{c} \in \mathbb{Z}^m$
         $p_k$: the $k$-th component of $\vec{p} \in \mathbb{Z}^m$
         $A(N_1,\ldots,N_{m-1},*)$: an $m$-D array with $@A$ as its base address
         $\mathbb{E}$: the number of bytes per element of $A$ such that $\mathbb{E}$ divides $\mathbb{L}$
OUTPUT:  $\mathsf{MRPA}_{R,R'}:\mathcal{M}_R \to \mathcal{M}_{R'} \cup \{\perp\}$, i.e., $\mathsf{MRPA}_{R,R'}(\vec{a})$, where $\vec{a}=(\ell_1,i_1,\ldots,\ell_n,i_n,\alpha_{\vec{L}}(R))$
if ($m=1 \ \vee\ (@A \bmod \mathbb{L}=0 \ \wedge\ N_1 \bmod (\mathbb{L}\div\mathbb{E})=0)$)
    Set $C=1$
else
    Set $C=\lfloor(\mathbb{L}\div\mathbb{E}-2)/N_1\rfloor+2$      // typically $C=2$
$S_{R,R'}=\emptyset$
if ($\mathrm{rank}(H)=n$)
    for $c=-C+1,C-1$
        for $\ell=-\mathbb{L}\div\mathbb{E}+1,\mathbb{L}\div\mathbb{E}-1$
            Let $\mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})=\{\vec{b}\in\mathcal{M}_{R'} \mid \ \vec{b}\prec\vec{a}, H(\vec{a}_I-\vec{b}_I)=\vec{p}-\vec{c}+(\ell-cN_1,c,0,\ldots,0)\ \}$
            if ($H\vec{x}=\vec{p}-\vec{c}+(\ell-cN_1,c,0,\ldots,0)$ has an integer solution, say, $\vec{g}\in\mathbb{Z}^n$)
                Set $\vec{b}^*_I=(i_1-g_1,\ldots,i_n-g_n)$
                if ($\vec{b}^*\prec\vec{a}$)
                    $S_{R,R'}=S_{R,R'}+\{\vec{b}^*\}$
else if ($H$ is uncoupled (Def. 11) and $RIS_{R'}$ is rectangular)
    Let $RIS_{R'}=\{(j_1,\ldots,j_n)\mid \forall\ 1\leqslant k\leqslant n: L'_k\leqslant j_k\leqslant U'_k\}$
    Let $R$ and $R'$ be enclosed in $d$ common loops, where $0\leqslant d\leqslant n$
    for $c=-C+1,C-1$
        for $\ell=-\mathbb{L}\div\mathbb{E}+1,\mathbb{L}\div\mathbb{E}-1$
            for $z=1,d+1$
                Let $\mathcal{Q}^{\ell,c,z}_{R,R'}(\vec{a})=\left\{\vec{b}\in\mathcal{M}_{R'}\ \middle|\ \begin{array}{l}\vec{b}=(\ell'_1,j_1,\ldots,\ell'_n,j_n,\alpha_{\vec{L}'}(R'))\\ ((j_1,\ldots,j_d),b_{2d+1})\prec_z((i_1,\ldots,i_d),a_{2d+1})\\ H(\vec{a}_I-\vec{b}_I)=\vec{p}-\vec{c}+(\ell-cN_1,c,0,\ldots,0)\end{array}\right\}$
                if (the system after the '|' in terms of the variables $j_1,\ldots,j_n$ is consistent)
                    Set $(j^*_1,\ldots,j^*_{z-1})=(i_1,\ldots,i_{z-1})$
                    if ($z\neq d+1$)
                        Set $j^*_z$ to its unique solution if it has one and $\min(i_z-1,U'_z)$ otherwise
                    for $k=z,n$
                        if ($z=d+1\vee k>z$)
                            Set $j^*_k$ to its unique solution if it has one and $U'_k$ otherwise
                    Set $\vec{b}^*_I=(j^*_1,\ldots,j^*_n)$
                    $S_{R,R'}=S_{R,R'}+\{\vec{b}^*\}$
else
    for $c=-C+1,C-1$
        for $\ell=-\mathbb{L}\div\mathbb{E}+1,\mathbb{L}\div\mathbb{E}-1$
            Let $\mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})=\left\{\vec{b}\ \middle|\ \begin{array}{l}\vec{b}\prec\vec{a},\vec{b}\in\mathcal{M}_{R'},\vec{a}\in\mathcal{M}_R\\ H(\vec{a}_I-\vec{b}_I)=\vec{p}-\vec{c}+(\ell-cN_1,c,0,\ldots,0)\end{array}\right\}$
            Let $\vec{b}^*$ be its lexicographic maximum (Section 4.3.1)
            $S_{R,R'}=S_{R,R'}+\{\vec{b}^*\}$
$\mathsf{MRPA}_{R,R'}(\vec{a})=\max_{\prec}\{\vec{b}^*\in S_{R,R'}\mid \vec{b}^*\in\mathcal{M}_{R'}, ml_R(\vec{a})=ml_{R'}(\vec{b}^*)\}$
```

Fig. 5. An algorithm for computing $\mathsf{MRPA}_{R,R'}$ when R and R' are uniformly generated.

distances from $\vec{a}$. We consider only distances up to the line size as beyond which we can never access the same memory line. There are a total of $2\mathbb{L} \div \mathbb{E} - 1$ such sets:

$$\mathcal{Q}^{\ell}_{R,R'}(\vec{a}) \quad = \quad \{\vec{b} \in \mathcal{M}_{R'} \mid \vec{b} \prec \vec{a}, ma_R(\vec{a}) - ma_{R'}(\vec{b}) = \ell\} \tag{12}$$

where $|\ell| < \mathbb{L} \div \mathbb{E}$. The following two conclusions are a direct consequence of this definition:

- All accesses in the set $\mathcal{Q}^{\ell}_{R,R'}(\vec{a})$ are mapped to the same memory line as $\vec{a}$ or all are not:

$$\forall \vec{b}_1, \vec{b}_2 \in \mathcal{Q}^{\ell}_{R,R'}(\vec{a}) : ml_R(\vec{a}) = ml_{R'}(\vec{b}_1) \iff ml_R(\vec{a}) = ml_{R'}(\vec{b}_2) \tag{13}$$

- $\mathcal{P}_{R,R'}(\vec{a}) \subseteq \cup_{|\ell|<\mathbb{L}\div\mathbb{E}} \mathcal{Q}^{\ell}_{R,R'}(\vec{a})$.

Hence, the function $ipred_{R,R'}$ can also be obtained precisely as follows:

$$ipred_{R,R'}(\vec{a}) \quad = \quad \max_{\prec}\{\vec{b} \in O_{R,R'} \mid ml_R(\vec{a}) = ml_{R'}(\vec{b})\} \tag{14}$$

where

$$O_{R,R'} \quad = \quad \{\max_{\prec} Q^{\ell}_{R,R'}(\vec{a}) \mid |\ell| < \mathbb{L} \div \mathbb{E}\,\} \tag{15}$$

This development is significant. We are only required to find $\max_{\prec} \mathcal{Q}^{\ell}_{R,R'}(\vec{a})$ during *reuse analysis* by working with the linear constraint $ma_R(\vec{a}) - ma_{R'}(\vec{b}) = \ell$ that appears in (12). The non-linear constraint $ml_R(\vec{a}) = ml_{R'}(\vec{b})$ that appears in (14) will be checked during *behavior analysis* when $\vec{a}$ is analyzed and is thus known in numbers.

To find $\max_{\prec} \mathcal{Q}^{\ell}_{R,R'}(\vec{a})$, we divide $\mathcal{Q}^{\ell}_{R,R'}(\vec{a})$ further — this is where our approximation arises. In lines 10 – 13, C is found to be the maximum number of array columns possibly spanned by a memory line. In line 11, C is set to 1 if A is a 1-D array or its columns are aligned to the memory line boundaries. In line 13, C is typically 2 since $\mathbb{L} \div \mathbb{E} - 2 < N_1$. (Recall that $\vec{a}_I$ and $\vec{b}_I$ identify the iterations at which the accesses $\vec{a}$ and $\vec{b}$ are executed.) Based on the notations in lines 2 – 9, we find that

$$\begin{aligned} ma_R(\vec{a}) &= @A + \textstyle\sum_{i=1}^{m}(\Pi_{k=1}^{i-1}N_k)(\vec{h}_i\vec{a}_I + c_i - 1) \\ ma_{R'}(\vec{b}) &= @A + \textstyle\sum_{i=1}^{m}(\Pi_{k=1}^{i-1}N_k)(\vec{h}_i\vec{b}_I + p_i - 1) \end{aligned} \tag{16}$$

Thus, we can rewrite $ma_R(\vec{a}) - ma_{R'}(\vec{b}) = \ell$ to:

$$\sum_{i=1}^{m}(\Pi_{k=1}^{i-1}N_k)(\vec{h}_i(\vec{a}_I - \vec{b}_I) + c_i - p_i) \quad = \quad \ell \tag{17}$$

The base address $@A$ that got cancelled out in the above formula is accounted for in the constraint $ml_R(\vec{a}) = ml_{R'}(\vec{b})$ introduced in line 45. In practice, it is rare for $\vec{a}$ and $\vec{b}$ to *touch* the same memory line even when the third indices (if any) of the accessed elements differ. Thus, we impose the following restrictions on the last $m-1$ dimensions of A:

$$\begin{array}{rcl} \vec{h}_2(\vec{a}_I - \vec{b}_I) + c_2 - p_2 & = & c \\ \vec{h}_3(\vec{a}_I - \vec{b}_I) + c_3 - p_3 & = & 0 \\ & \cdots & \\ \vec{h}_m(\vec{a}_I - \vec{b}_I) + c_m - p_m & = & 0 \end{array} \tag{18}$$

where $|c| < C$. This implies that the two elements accessed by $\vec{a}$ and $\vec{b}$ share the identical indices in their last $m-2$ dimensions. The number of the memory lines accessed this way, i.e., spanned by the first two dimensions of A, is $O((\Pi_{i=1}^{m} N_i)/(\mathbb{L} \div \mathbb{E}))$. The situation when, for example, $A(N_1, N_2, 1, \ldots, 1)$ and $A(1, 1, 2, 1, \ldots, 1)$ are the two accessed elements in a memory line can be analyzed similarly by using different constant terms in (18). The memory lines of this second kind are totaled as $O(\Pi_{i=3}^{m} N_i)$ and are thus negligible.

By combining (17) and (18), we obtain:

$$H(\vec{a}_I - \vec{b}_I) \;=\; \vec{p} - \vec{c} + (\ell - cN_1, c, 0, \ldots, 0) \tag{19}$$

Thus, we have found a desirable decomposition of $\mathcal{Q}^{\ell}_{R,R'}(\vec{a})$ as follows:

$$\mathcal{Q}^{\ell}_{R,R'}(\vec{a}) \;=\; (\bigcup_{|c|<C} \mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})) \cup \mathcal{Q}^{\ell}_{other}(\vec{a}) \tag{20}$$

where $\mathcal{Q}^{\ell}_{other}(\vec{a})$ is ignored in our current implementation and $\mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})$ is defined by:

$$\mathcal{Q}^{\ell,c}_{R,R'}(\vec{a}) \;=\; \left\{ \vec{b} \in \mathcal{M}_{R'} \;\middle|\; \begin{array}{l} \vec{b} \prec \vec{a} \\ H(\vec{a}_I - \vec{b}_I) = \vec{p} - \vec{c} + (\ell - cN_1, c, 0, \ldots, 0) \end{array} \right\} \tag{21}$$

By combining all the results so far, the function $\mathsf{MRPA}_{R,R'}$ is constructed as follows:

$$\mathsf{MRPA}_{R,R'}(\vec{a}) \;=\; \max_{\prec}\{\vec{b} \in S_{R,R'} \mid ml_R(\vec{a}) = ml_{R'}(\vec{b})\} \tag{22}$$

where

$$S_{R,R'} \;=\; \{\max_{\prec} \mathcal{Q}^{\ell,c}_{R,R'}(\vec{a}) \mid |\ell| < \mathbb{L} \div \mathbb{E}, |c| < C\} \tag{23}$$

FindMRPA distinguishes the three cases in lines 15, 23 and 39. In each case, the c and ℓ loops are responsible for computing $\max_{\prec} \mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})$ exactly for all possible pairs (ℓ, c).

• **The** if **Case.** $\mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})$ contains the singleton $\vec{b}^*$ found in line 20 if $\vec{b}^* \in \mathcal{M}_{R'}$ and is empty otherwise. Thus, $\max_\prec \mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})$ is $\vec{b}^*$ if $\vec{b}^* \in \mathcal{M}_{R'}$ and $\perp$ otherwise. This explains the occurrence of $\vec{b}^* \in \mathcal{M}_{R'}$ in line 45. This case takes $O((\mathbb{L} \div \mathbb{E}) \times C \times \max(n,m)^3)$.

• **The "**else if**" Case.** This case is less expensive because H has a simpler structure. In line 25, $(\ell_1, \ldots, \ell_d) = (\ell'_1, \ldots, \ell'_d)$ holds. We further divide $Q^{\ell,c}_{R,R'}(\vec{a})$ given in (21) such that $\mathcal{Q}^{\ell,c}_{R,R'}(\vec{a}) = \cup_{z=1}^{d+1} \mathcal{Q}^{\ell,c,z}_{R,R'}(\vec{a})$, where $\mathcal{Q}^{\ell,c,z}_{R,R'}(\vec{a})$ is defined in line 29. This is because $\vec{b} \prec \vec{a} \equiv \vee_{k=1}^{d+1}((b_2, \ldots, b_{2d}), b_{2d+1}) \prec_k ((a_2, \ldots, a_{2d}), a_{2d+1}) \equiv \vee_{k=1}^{d+1}((j_1, \ldots, j_d), b_{2d+1}) \prec_k ((i_1, \ldots, i_d), a_{2d+1})$, where $\prec_k$ is from (9). Note that $\max_\prec \mathcal{Q}^{\ell,c,z}_{R,R'}(\vec{a})$ is $\vec{b}^*$ if $\vec{b}^* \in \mathcal{M}_{R'}$ and $\perp$ otherwise. Hence, $\vec{b}^* \in \mathcal{M}_{R'}$ in line 45. Clearly, we have:

$$\max_\prec \mathcal{Q}^{\ell,c}_{R,R'}(\vec{a}) \quad = \quad \max_\prec\{\max_\prec \mathcal{Q}^{\ell,c,z}_{R,R'}(\vec{a}) \mid 1 \leqslant z \leqslant d+1\} \tag{24}$$

• **The** else **Case.** This requires integer programming but is invoked only rarely. In addition, the problem in line 42 can generally be solved more efficiently than (8) since the constraint variables in the last constraint have smaller coefficients than those in the last three of (8). Note that $\vec{a} \in \mathcal{M}_R$ appears in line 42 and $\vec{b} \in \mathcal{M}'_R$ is redundant in line 45.

Example 4 (The if **Case).** *Let us construct* $\mathsf{MRPA}_{B(I_2,I_1),B(I_2-1,I_1)}$ *for Figure 4, where* $\vec{a} = (1, i_1, 2, i_2, 1)$. *Suppose* $\mathbb{L} = 32$ *and* $@B = 3200$. *Thus,* $\mathbb{L} \div \mathbb{E} = 4$. *In line 11, we set* $C = 1$. $H = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ *is nonsingular,* $\vec{c} = (0,0)$ *and* $\vec{p} = (-1,0)$. *The* if *case is executed. In line 19, we solve* $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]\vec{x} = \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right] + \left[\begin{smallmatrix} \ell \\ 0 \end{smallmatrix}\right]$. *The unique solution found in line 20,* $\vec{b}^* = (1, i_1, 1, i_2 - \ell + 1, 1)$, *satisfies* $\vec{b}^* \prec \vec{a}$. *Finally,* $S_{B(I_2,I_1),B(I_2-1,I_1)} = \{(1, i_1, 1, i_2 - \ell + 1, 1) \mid -3 \leqslant \ell \leqslant 3\}$. *Thus,*

$$\mathsf{MRPA}_{B(I_2,I_1),B(I_2-1,I_1)}(\begin{bmatrix} 1 \\ i_1 \\ 2 \\ i_2 \\ 1 \end{bmatrix}) \quad = \quad \max_\prec \left\{ \begin{bmatrix} 1 \\ i_1 \\ 1 \\ i_2 - \ell + 1 \\ 1 \end{bmatrix} \middle| \begin{array}{l} -3 \leqslant \ell \leqslant 3 \\ 1 \leqslant i_1 \leqslant 20 \\ 2 \leqslant i_2 - \ell + 1 \leqslant 20 \\ \lfloor \frac{i_2 - 1 + 20(i_1 - 1)}{4} \rfloor = \lfloor \frac{i_2 - \ell - 1 + 20(i_1 - 1)}{4} \rfloor \end{array} \right\}$$

Example 5 (The "else if**" Case.).** *Let us construct* $\mathsf{MRPA}_{X(k,i),X(k,i)}$ *for the MM kernel given in Appendix B, where* $\vec{a} = (1, i_1, 1, i_2, 1, j_3, 2) \triangleq (1, i, 1, j, 1, k, 2)$. *Thus, the single nest in the kernel is identified by the loop vector* $(1,1,1)$, *where* $\alpha_{(1,1,1)}(X(k,i)) = 2$. X *has the declaration* $X(N,N)$. *Suppose* $\mathbb{L} = 32$, $@X = 3224$ *and* $N = 100$. *Thus,* $\mathbb{L} \div \mathbb{E} = 4$. $H = \left[\begin{smallmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{smallmatrix}\right]$ *and* $\vec{c} = \vec{p} = (0,0)$. *Note that* $\operatorname{rank}(H) < n = 3$. $X(k,i)$ *is uncoupled. Its RIS is cubic:* $RIS_{X(k,i)} = \{(i,j,k) \mid 1 \leqslant i,j,k \leqslant N\}$. *Thus, the "*else if*" case is executed. In line 13, we set* $C = 2$. $X(k,i)$ *is enclosed in all the three loops. So in line 25,*

$d = 3$. In line 29, we have $\mathcal{Q}^{\ell,c,z} = \{(1, j_1, 1, j_2, 1, j_3, 2) \mid 1 \leqslant j_1, j_2, j_3 \leqslant N, (j_1, j_2, j_3, 2) \prec_z (i_1, i_2, i_3, 2), i_3 - j_3 = \ell - cN_1, i_1 - j_1 = c\}$ with the subscripts "$X(k,i), X(k,i)$" and the argument $\vec{a} = (1, i_1, 1, i_2, 1, i_3, 2)$ for $\mathcal{Q}$ ignored to avoid cluttering. The c loop in line 26 has three iterations. In the first iteration when $c = -1$, $i_1 - j_1 = -1$ and $(j_1, j_2, j_3, 2) \prec_z (i_1, i_2, i_3, 2)$ are found to be inconsistent for all ℓ and z in line 30. That is, $\mathcal{Q}^{\ell,-1,z} = \emptyset$ for all ℓ and z. In the second iteration when $c = 0$, there are two cases. Case (I) $-3 \leqslant \ell \leqslant 0$. We have $\mathcal{Q}^{\ell,0,1} = \mathcal{Q}^{\ell,0,3} = \mathcal{Q}^{\ell,0,4} = \emptyset$. From $\mathcal{Q}^{\ell,0,2}$, we obtain $\vec{b}^*_I = (i_1, \min(i_2 - 1, N), i_3 - \ell) = (i_1, i_2 - 1, i_3 - \ell)$ and add the elements in $S_1 = \{(1, i_1, 1, i_2 - 1, 1, i_3 - \ell, 2) \mid -3 \leqslant \ell \leqslant 0\}$ to $S_{X(k,i),X(k,i)}$. Case (II) $0 < \ell \leqslant 3$. We have $\mathcal{Q}^{\ell,0,1} = \mathcal{Q}^{\ell,0,4} = \emptyset$. From $\mathcal{Q}^{\ell,0,2}$, we obtain $\vec{b}^*_I = (i_1, \min(i_2 - 1, N), i_3 - \ell) = (i_1, i_2 - 1, i_3 - \ell)$ and add what are in $S_2 = \{(1, i_1, 1, i_2 - 1, 1, i_3 - \ell, 2) \mid 0 < \ell \leqslant 3\}$ to $S_{X(k,i),X(k,i)}$. From $\mathcal{Q}^{\ell,0,3}$, we obtain $\vec{b}^*_I = (i_1, i_2, i_3 - \ell)$ and add the elements in $S_3 = \{(1, i_1, 1, i_2, 1, i_3 - \ell, 2) \mid 0 < \ell \leqslant 3\}$ to $S_{X(k,i),X(k,i)}$. In the last iteration when $c = 1$, we have $\mathcal{Q}^{\ell,1,2} = \mathcal{Q}^{\ell,1,3} = \mathcal{Q}^{\ell,1,4} = \emptyset$. From $\mathcal{Q}^{\ell,1,1}$, we obtain $\vec{b}^*_I = (i_1 - 1, N, i_3 + N - \ell)$ and add the elements in $S_4 = \{(1, i_1 - 1, 1, N, 1, i_3 + N - \ell, 2) \mid -3 \leqslant \ell \leqslant 3\}$ to $S_{X(k,i),X(k,i)}$. Finally, $S_{X(k,i),X(k,i)} = S_1 \cup S_2 \cup S_3 \cup S_4$. The function $\mathsf{MRPA}_{X(k,i),X(k,i)}$ is given as in line 45. Appendix A discusses how to remove some redundant elements from $S_{X(k,i),X(k,i)}$ for this example.

Example 6 (The else Case). *Let us construct $\mathsf{MRPA}_{D(I_2),D(I_2-1)}$ for the same reference pair in Example 3. Since $H = (0, 1)$, we have $\mathrm{rank}(H) \neq n$. $RIS_{D(I_2-1)}$ is non-rectangular. Hence, the else case is executed. In line 11, we set $C = 1$ since C is one-dimensional. Replacing the last three constraints in (10) with $i_2 - j_2 = \ell - 1$ and solving the resulting system at every iteration of the ℓ loop in line 41, we obtain $S_{D(I_2),D(I_2-1)} = \{(1, 20, 1, i_2 - \ell, 2) \mid -3 \leqslant \ell \leqslant 3\}$. The resulting $\mathsf{MRPA}_{D(I_2),D(I_2-1)}$ is equivalent to $ipred_{D(I_2),D(I_2-1)}$ found in Example 3.*

$\mathsf{MRPA}_{R,R'}$ is exact in the common cases when A is a 1-D or 2-D array or when A is any-dimensional array provided its columns are aligned to the memory line boundaries.

Theorem 3: *If $1 \leqslant m \leqslant 2$ or $@A \bmod \mathbb{L} = 0 \wedge N_1 \bmod (\mathbb{L} \div \mathbb{E}) = 0$, $\mathsf{MRPA}_{R,R'} = ipred_{R,R'}$.*

Proof: Under the hypothesis, $\mathcal{Q}^{\ell}_{other}(\vec{a}) = \emptyset$ in (20). In addition, *FindMRPA* builds $\max_{\prec} \mathcal{Q}^{\ell,c}_{R,R'}(\vec{a})$ exactly. By noting the definition of $ipred_{R,R'}$ in (14) and (15) and the definition of $\mathsf{MRPA}_{R,R'}$ in (22) and (23), we conclude that $\mathsf{MRPA}_{R,R'} = ipred_{R,R'}$. ■

The MRPA functions found in Examples 4 – 6 are all exact.

In our current implementation, R and R' are also considered uniformly generated if the arrays accessed are aliases created by COMMON or EQUIVALENCE statements provided that both have the same dimension, shape and extent. Thus,

$$\mathsf{MRPA}_R(\vec{a}) = \max_{\prec}\{\mathsf{MRPA}_{R,R'}(\vec{a}) \mid R' \in \mathit{RefSet}\} \tag{25}$$

4.3.3 Efficiency and Accuracy Trade-offs

When building $\mathsf{MRPA}_{R,R'}$, we have used a strict subset of $\mathcal{M}_{R'}$ if $\mathcal{Q}^{\ell}_{other}(\vec{a}) \neq \emptyset$ in (20). When building MRPA_R, we have considered only the accesses of all R' references that are uniformly generated w.r.t R in (25). These facts lead directly to the following result.

Theorem 4: $\mathsf{MRPA}_{R,R'} \preceq \mathit{ipred}_{R,R'}$ *and* $\mathsf{MRPA}_R \preceq \mathit{ipred}_R$.

We have discussed how to construct ipred_R exactly and MRPA_R efficiently for numerical codes. Our reuse framework is flexible and extensible, allowing various 'analysis' switches to be used. Several possibilities are: (a) including $\mathcal{Q}^{\ell}_{other}(\vec{a})$ given in (20) in reuse analysis, (b) including non-rectangular shapes for $\mathit{RIS}_{R'}$ in line 23, (c) analyzing non-uniformly generated references and (d) analyzing aliased arrays with, for example, different dimensions. These switches allow various trade-offs between efficiency and accuracy to be made. For example, time-consuming deep analysis is required in order to optimize a program's hot spots.

4.3.4 MRPAs v.s. Shortest Reuse Vectors

Given $\vec{a} \in \mathcal{M}_R$ and $\vec{b} \in \bigcup_{R' \in \mathit{RefSet}} \mathcal{P}_{R,R'}(\vec{a})$, $\vec{a} - \vec{b}$ represents a generalization of Wolf and Lam's reuse vector from single nests to multiple nests [57]. Among all these reuse vectors of the access $\vec{a}$, the *shortest reuse vector* (SRV) in the entire program is:

$$\mathsf{SRV}_R(\vec{a}) \;\; = \;\; \vec{a} - \mathit{ipred}_R(\vec{a}) \;\succ\; \vec{0} \tag{26}$$

(with the provision that $\vec{a} - \perp = \top$ and $\top \succ$ any reuse vector.) Accordingly, $\vec{a} - \mathsf{MRPA}_R(\vec{a})$ is our approximation of $\mathsf{SRV}_R(\vec{a})$. To apply Theorem 1 to check if the access $\vec{a}$ is a hit or a miss, we need its shortest reuse vector $\mathsf{SRV}_R(\vec{a})$ in order to obtain the MRPA, $\vec{a} - \mathsf{SRV}_R(\vec{a})$. Thus, the concepts of MRPAs and shortest reuse vector are equivalent but the latter represents a roundabout way of providing what is required in Theorem 1.

As can be observed from Examples 4 – 6, $ipred_{R,R'}$ depends on the following factors: (a) the subscript expressions of R and R', (b) the base addresses of the arrays accessed by R and R', (c) the array sizes in all but the last dimension, (d) the line size $\mathbb{L}$, (e) the shape of RIS_R and (f) the shape of $RIS_{R'}$. *FindMRPA*, which takes all these factors into account, is exact in the common cases described in Theorem 3. Wolf and Lam's reuse framework for single nests [57], while sufficient for their optimization purposes, does not provide the notion of shortest reuse vector required in behavior analysis. This is because they compute reuse vectors without considering the factors (b) – (f) above. For example, they would have generated only $(0, 1, 0)$ and $(0, 0, 1)$ for Example 5. Ghosh *et al* [22] rely on Wolf and Lam's framework and some ad hoc techniques to obtain approximately the shortest reuse vectors required in their CMEs for single perfect nests in the absence of IF conditionals.

5 Cache Behavior Modeling

A program's cache behavior is specified based on the notion of MRPA. An algorithm for obtaining predictions of miss ratios from this specification is given.

5.1 Cache Miss Specification

The following result is a direct consequence of Theorem 1(a).

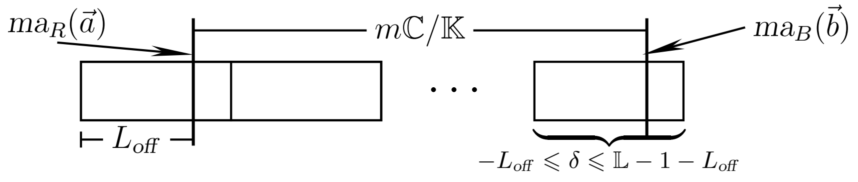
Theorem 5: *The access $\vec{a}$ of R is a cold miss if* $\mathsf{MRPA}_R(\vec{a}) = \perp$.

If $\mathsf{MRPA}_R(\vec{a}) \neq \perp$, we follow the CMEs [22] to set up the so-called replacement miss equations to investigate if the access $\vec{a}$ is a hit or a miss. Let $InterRefs_R(\vec{a})$ be the set of all references that may be potentially accessed between the access $\mathsf{MRPA}_R(\vec{a})$ and the access $\vec{a}$. The construction of this set is straightforward and is omitted here.

The replacement miss equations for the access $\vec{a}$ of R are as follows:

$$\mathsf{RME}_R(\vec{a}) \quad = \quad \begin{cases} \bigvee_{B \in InterRefs_R(\vec{a})} \\ \qquad \big(\vec{b} \triangleq (\ell_1^i, j_1, \ldots, \ell_n^i, j_n, \alpha_{(\ell_1^i,\ldots,\ell_n^i)}(B)) \in \mathcal{M}_B \\ \qquad \wedge\ \mathsf{MRPA}_R(\vec{a}) \prec \vec{b} \prec \vec{a} \\ \qquad \wedge\ ma_R(\vec{a}) + m\mathbb{C}/\mathbb{K} + \delta = ma_B(\vec{b}) \\ \qquad \wedge\ L_{\mathit{off}} \triangleq ma_R(\vec{a}) \bmod \mathbb{L} \\ \qquad \wedge\ -L_{\mathit{off}} \leqslant \delta \leqslant \mathbb{L} - 1 - L_{\mathit{off}} \\ \qquad \wedge\ m \neq 0\big) \end{cases} \tag{27}$$

The last four lines due to [22] are equivalent to $cs_R(\vec{a}) = cs_B(\vec{b}) \wedge ml_R(\vec{a}) \neq ml_B(\vec{b})$ except



$ma_R(\vec{a})$
$m\mathbb{C}/\mathbb{K}$
$ma_B(\vec{b})$
L_{off}
$-L_{\text{off}} \leqslant \delta \leqslant \mathbb{L} - 1 - L_{\text{off}}$

Fig. 6. The last four lines in (27) for checking cache set contentions [22]. The boxes depict memory lines.

they are expressed in terms of bytes here. They are illustrated in Figure 6 in order to make the paper self-contained, where $ma_B(\vec{b})$ is at a distance of $m\mathbb{C}/\mathbb{K} + \delta$ bytes in memory from $ma_R(\vec{a})$. Note that $m \neq 0$ ensures that $ml_R(\vec{a}) \neq ml_B(\vec{b})$. In a direct-mapped cache when both R and B take $\mathbb{L}$ bytes to store a single array element, we can set $\delta = 0$.

The replacement miss equations are to analyze if the access $\vec{a}$ can reuse the memory line $ml_R(\vec{a})$ that was most recently accessed by the access $\mathsf{MRPA}_R(\vec{a})$ subject to the set contentions caused by all intervening accesses of the references in $\mathit{InterRefs}_R(\vec{a})$. A *set contention* is said to occur when one such an intervening access $\vec{b}$ is made to a memory line that is distinct from $ml_R(\vec{a})$ but also mapped to $cs_R(\vec{a})$ (enforced by the last four lines of (27)). The intervening access $\vec{b}$ causes a set contention if the associated m is a solution to (27). Hence, the following result is a direct consequence of Theorem 1(b).

Theorem 6: *Suppose* $\mathsf{MRPA}_R(\vec{a}) \neq \perp$. *The access* $\vec{a}$ *of* R *is a replacement miss if (27) has at least* $\mathbb{K}$ *distinct solutions* $m_1, \ldots, m_{\mathbb{K}}$ *and a hit otherwise.*

In comparison with the CMEs [22], we have dispensed with their cold miss equations and generalized their replacement miss equations for single nests to multiple nests.

5.2 Cache Miss Generation

EstimateMisses given in Figure 7 finds the miss ratios for all references independently and derives the miss ratio for the program from these individual miss ratios. This algorithm analyzes a sample of RIS_R based on standard statistical sampling techniques. For technical details regarding c and w, see [11]. Basically, these two input values determine the size of the sample taken from RIS_R and impose a lower bound on $|\mathit{RIS}_R|$. Their meanings are such that if we run *EstimateMisses* many times, the real miss ratio for each R obtained in c of these runs will lie in the interval $[\mathit{MissRatio}(R) - w/2, \mathit{MissRatio}(R) + w/2]$. However, this interpretation does not apply to the miss ratio for the entire program.

Our sampling mechanism is based on the prior work [53] except that we have made

```
1  Algorithm EstimateMisses
2  INPUT:    $c$: the confidence percentage
             $w$: the confidence interval
3  OUTPUT: ProgMissRatio
4  for each reference $R$ (in no particular order)
5      compute $|RIS_R|$, i.e., the volume of $RIS_R$
6      if ($|RIS_R|$ is too small to achieve $(c, w)$)
7          if ($|RIS_R|$ is large enough to achieve
8                  the default $(c', w') = (90\%, 0.15)$)
9              $\mathcal{S}(R)$ = a sample $(c', w')$ of $RIS_R$
10         else
11             $\mathcal{S}(R) = RIS_R$ // analyze all accesses
12     else
13         $\mathcal{S}(R)$ = a sample $(c, w)$ of $RIS_R$
14 for each reference $R$ in the program
15     $CM_R = \emptyset$        // cold misses for $R$
16     $RM_R = \emptyset$        // replacement misses for $R$
17     $H_R = \emptyset$         // hits for $R$
18     for each iteration vector $\vec{a}_I \in S(R)$
19         if ($\mathsf{MRPA}_R(\vec{a}) = \perp$) // Theorem 5
20             $CM_R = CM_R \cup \{\vec{a}\}$
21         else
22             if ($\vec{a}$ is a repl. miss by Theorem 6)
23                 $RM_R = RM_R \cup \{\vec{a}\}$
24             else
25                 $H_R = H_R \cup \{\vec{a}\}$
26     $MissRatio(R) = \frac{|CM_R|+|RM_R|}{|\mathcal{S}(R)|}$
27 $ProgMissRatio = \frac{\sum_R |RIS_R| \times MissRatio(R)}{\sum_R |RIS_R|}$
```

Fig. 7. An algorithm for estimating cache misses.

modifications on computing $|RIS_R|$ when RIS_R is not convex. As discussed in Section 4.2.2, RIS_R can always be expressed as a union of convex polytopes. We compute $|RIS_R|$ by slicing RIS_R recursively into regions of lower and lower dimensions until eventually every such a region is either empty or has a few line segments so that the points in the region can be counted easily. This algorithm, while exponential in terms of the dimensionality of RIS_R, is very efficient for regular codes with simple loop bounds and IF conditionals. The majority of RISs are rectangular; their volumes can be calculated trivially. Other methods for computing the volume of a convex polytope can be found in [9], [43].

The for loop in line 14 analyzes all references sequentially. For each reference R being analyzed, the for loop in line 18 goes through its access vectors in its sample and classifies each as a cold miss, replacement miss or a hit by applying Theorems 5 and 6. The time complexity for solving (27) in line 22 is $O(M)$, where M is the number of the intervening accesses investigated for the reuse. In practice, *EstimateMisses* is efficient due to sampling.

Our model, when mispredicting at an access, always errs on the conservative side due to Theorem 4. If $\mathsf{MRPA}_R(\vec{a}) = \perp$ but $ipred_R(\vec{a}) \neq \perp$, the access $\vec{a}$ may be either a replacement miss or a hit. Its classification as a (cold) miss in lines 19 – 20 is incorrect if it is actually a hit. If $\mathsf{MRPA}_R(\vec{a}) \prec ipred_R(\vec{a})$ but $\mathsf{MRPA}_R(\vec{a}) \neq \perp$, then the interval $(\mathsf{MRPA}_R(\vec{a}), \vec{a})$ strictly includes the interval $(ipred_R(\vec{a}), \vec{a})$. Hence, more memory accesses have been checked than necessary for set contentions due to the third line in (27). The access $\vec{a}$ is a hit when it is classified so in lines 21 – 25 but may still be a hit even when it is classified as a miss. These remarks highlight the importance of the notion of MRPA in exact behavior analysis.

6 Experiments

We have analyzed a range of programs from SPECfp95, Perfect Suite, Livermore Kernels, Linpack and Lapack. We discuss the experimental results for the five kernels given in Appendix B and three complete programs. Hydro_K consists of three 2-D loop nests from Livermore (kernel 18). MGRID_K is a 3-D loop nest (with multiple nests inside the outermost loop) from the MGRID Benchmark. MMT is the 3-D blocked loop nest from [20] for computing the matrix multiplication of A and B^T. MM is the matrix multiplication kernel with initialization taken from [25]. LWSI_K is a 4-D imperfect loop nest (with many statements between loops) from the LWSI Benchmark. These kernels (typical of nested regular computations) have been chosen so that all parts of our model can be exercised.

Let us recall the framework for exercising and validating our model given in Figure 1. In each program, scalars and temporaries are assumed register-allocated and each load/store array reference triggers a call to an appropriate instrumentation subroutine. All programs are compiled and linked using "g77 -O1" on a 933MHz Pentium III PC with 512MB memory. The base addresses and access order of all references are obtained from these binary codes. Hence, the miss ratio of a program is sensitive to a particular binary being used. However, the simulated code and analyzed code have the same accesses executed in the same order.

Our experimental results are summarized in a number of tables and figures. In each case, the problem size and the cache configurations used are indicated. The cache configurations covered include many from modern architectures such as Pentium IV (8KB, 64B, 4), UltraSparc III (64KB, 32B, 4), Alpha 21264 (64KB, 64B, 2), Athlon (64KB, 64B, 2) and PowerPC G4 (32KB, 32B, 8). All execution times are obtained on the 933MHz Pentium III PC. All simulation results are obtained using a trace-driven simulator.[8] In all experiments, we ran *EstimateMisses* with $c = 95\%$ and $w = 0.05$ as the input. The time elapsed on analyzing a program includes the costs from all the five components represented by the doubly-framed boxes in Figure 1.

[8]A locally written simulator has been used in all our experiments. It has been validated over the years against the well-known Dinero III trace-driven simulator [27].

6.1 Loop Nest Kernels

TABLE 2

Average Absolute Errors when Compared against Simulation for the Experiments from Figures 8 and 9.

Kernel	$\mathbb{K}$	(32KB,32B)	(8KB,64B)
Hydro_K	1	0.26	0.19
	2	0.25	0.18
	4	0.24	0.18
MGRID_K	1	0.20	0.35
	2	0.14	0.37
	4	0.14	0.30
MMT	1	0.43	0.19
	2	0.28	0.14
	4	0.21	0.09
MM	1	0.30	0.32
	2	0.27	0.30
	4	0.22	0.30
LWSI_K	1	0.57	0.99
	2	0.51	0.76
	4	0.53	0.61

Figure 8 compares the predicted miss ratios against that from simulation for a fixed (32KB, 32B) but with three different $\mathbb{K}$ choices. In each graph, the miss ratios are plotted against one single problem size parameter (with the others, if any, fixed). Figure 9 does the same for (8KB, 64B). Table 2 gives the average absolute errors. Table 3 shows the times taken to evaluate all the kernels for a single cache configuration. In all experiments, the predicted miss ratios are close to the simulated ones and are obtained in times that are at least two orders of magnitude faster than simulation. In Figures 8 and 9, the seemingly big differences between the predicted and simulated miss ratios for MGRID_K are due to the short ranges used for the miss ratios. We have used a smaller number of samples for MMT because the tile size parameters KN and JN are required to divide N.

6.2 Whole Programs

We evaluate *EstimateMisses* using three programs as detailed in Table 4. For each program, we succeeded in abstractly inlining all the calls and obtained an inlined program with loop nests only. Each program is analyzed using the *reference input data*. Thus, the variables in all READ statements are initialized from the reference data and then treated as compile-time constants. The three programs are further discussed below.

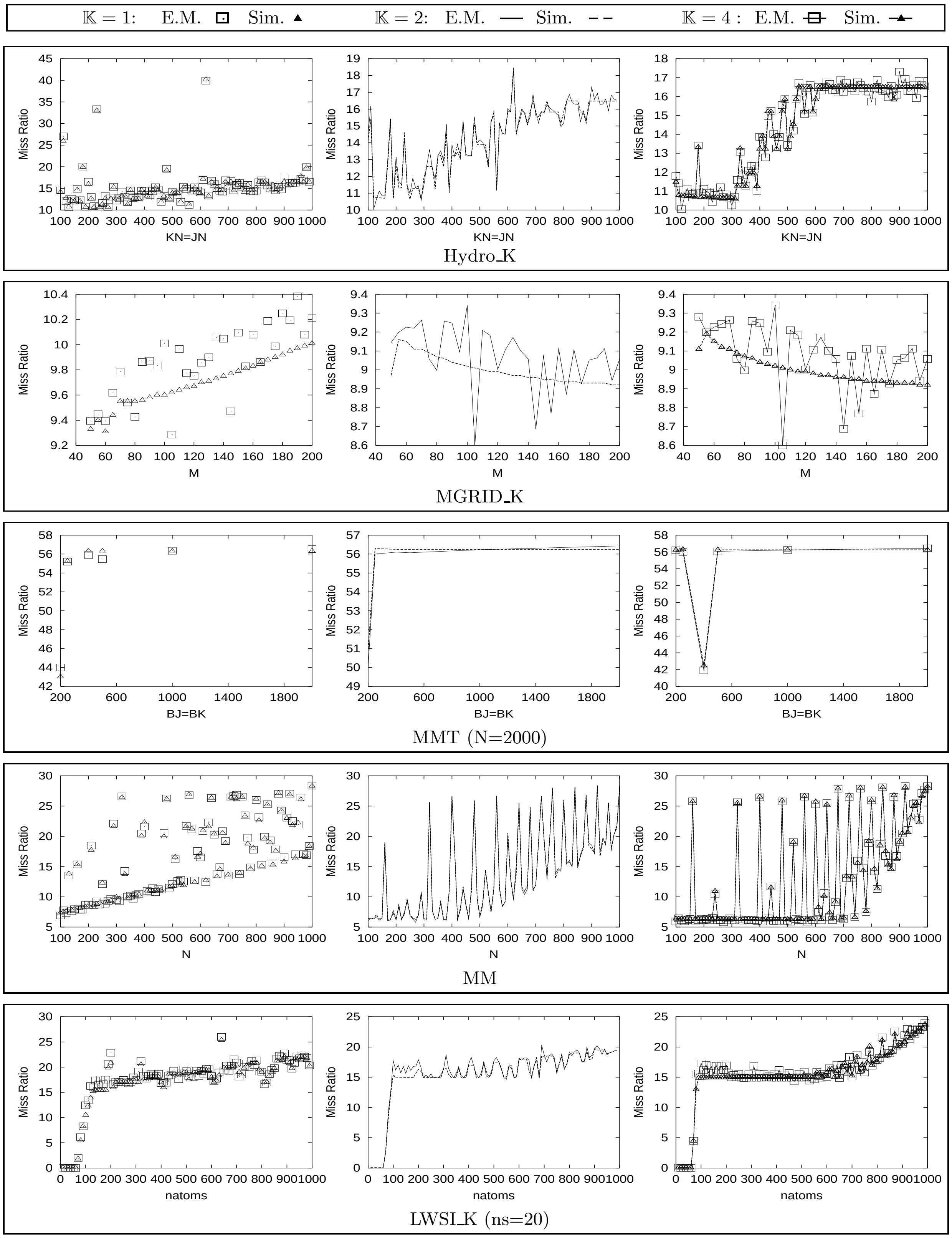


Fig. 8. Predicted and simulated miss ratios for $(\mathbb{C}, \mathbb{L})$ =(32KB, 32B) with three different associativities.

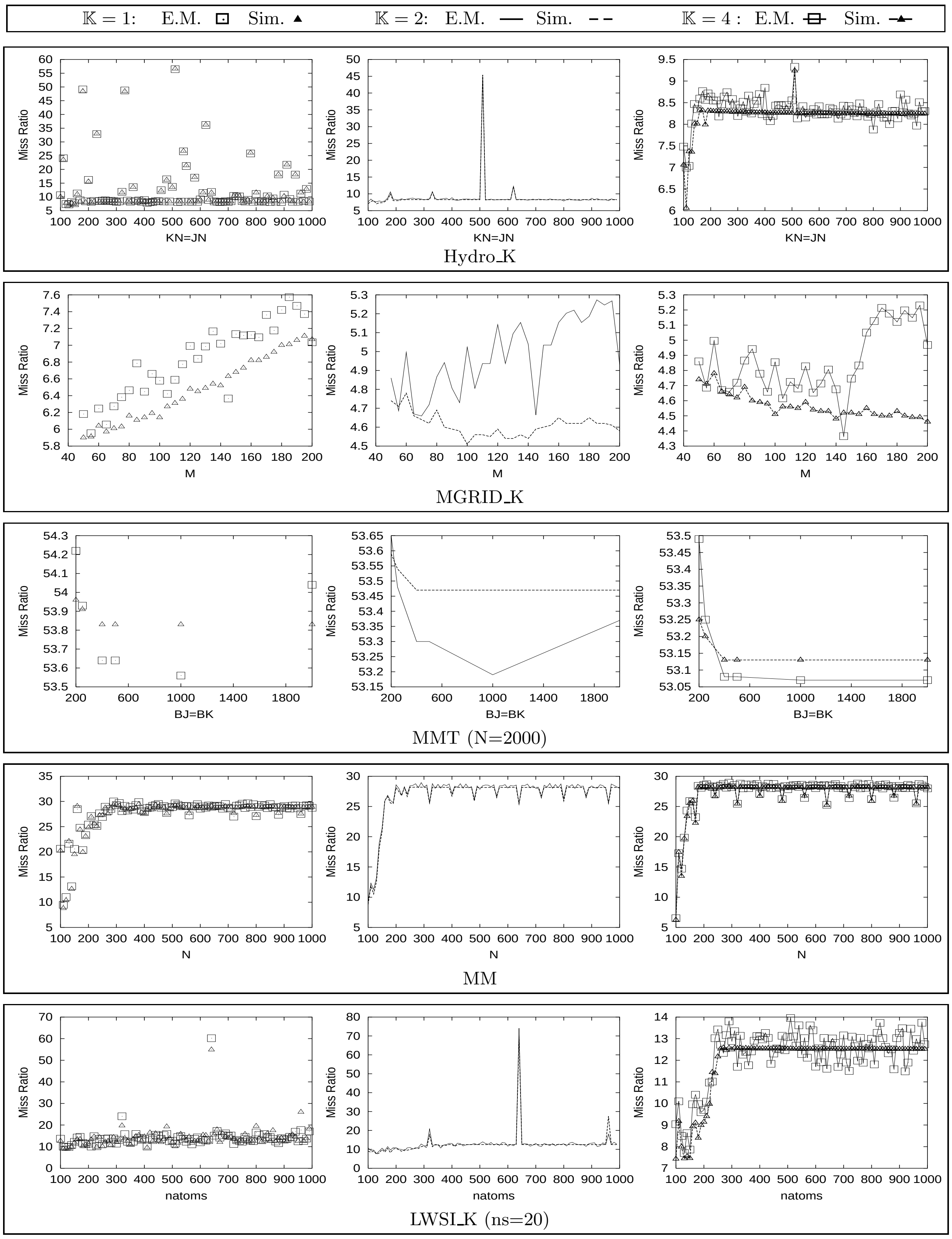


Fig. 9. Predicted and simulated miss ratios for $(\mathbb{C}, \mathbb{L})$ =(8KB, 64B) with three different associativities.

TABLE 3

Execution Times for (32KB,32B,2) Illustrated in Figure 8.

Kernel	E.M.(secs)			Sim.(secs)		
	Min.	Max.	Mean	Min.	Max.	Mean
Hydro_K	0.03	0.27	0.19	0.04	14.49	5.10
MGRID_K	0.17	0.19	0.18	0.50	35.00	11.82
MMT	0.71	15.54	4.71	1054.64	1938.65	1295.66
MM	0.07	0.89	0.33	0.13	1126.93	171.75
LWSI_K	0.06	0.28	0.17	0.21	18.33	8.20

TABLE 4

Three Complete Programs from SPECfp95.

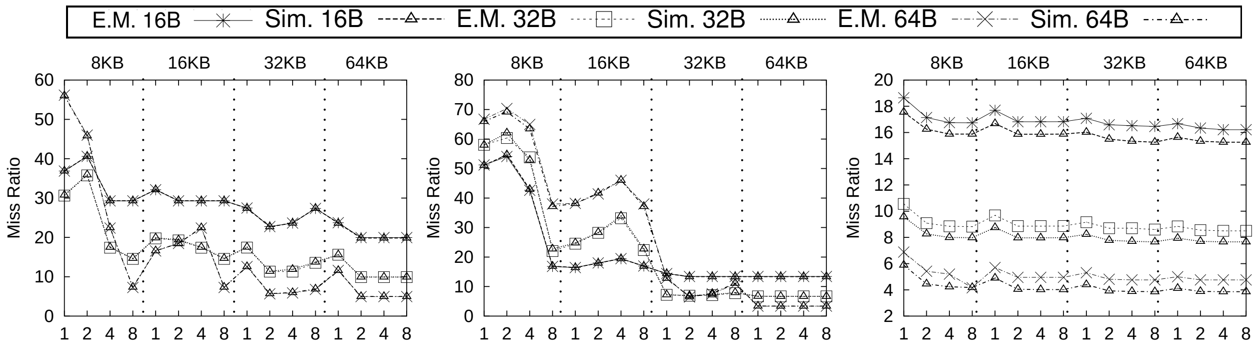
Program	#lines	#subrs	#calls	#refs
Tomcatv	190	1	0	79
Swim	429	6	6	52
Applu	3868	16	27	2565

• **Tomcatv.** This example shows that our model can analyze more than just isolated nests. The number of iterations of the outermost loop is data-dependent. For the reference input data, the outermost loop runs for 750 iterations. The only data-dependent IF conditional in the program is always false. In our analysis, the memory accesses contained in this conditional are included but those inside the IF body are ignored.

• **Swim.** This example shows that our model can analyze codes consisting of call statements. In this example, all calls are parameterless. After inlining, the outermost loop is an IF-GOTO construct, which is converted into a DO construct.

• **Applu.** This example shows that our model can analyze programs of this size efficiently with a good degree of accuracy. In subroutine SSOR, there are some data-dependent constructs. All but one are guarded by an IF conditional that is false at compile time and are thus ignored. The remaining one is a WRITE statement for a register-allocated scalar. The memory accesses in this IF conditional are included in our analysis.

Figure 10 compares the predicted miss ratios against that from simulation for $\mathbb{C}\times\mathbb{L}\times\mathbb{K} = 4 \times 3 \times 8 = 96$ cache configurations. The total analysis times required by Tomcatv, Swim and Applu for all the configurations lumped together are about 14 secs, 2.8 mins, 3.2 hours,



(a) Tomcatv (b) Swim (c) Applu

Fig. 10. Predicted and simulated miss ratios for $\mathbb{C} \times \mathbb{L} \times \mathbb{K} = 96$ cache configurations.

TABLE 5

Absolute Errors and Execution Times Compared Against Simulation for (8KB, 64B) from Figure 10.

Program	$\mathbb{K}$	Miss Ratio			Exe.T (secs)	
		E.M.	Sim.	Abs.Err.	E.M.	Sim.
Tomcatv	1	56.16	56.00	0.16	0.31	4780.48
	2	46.01	45.83	0.18	0.49	4897.58
	4	22.38	22.46	0.08	0.60	4878.56
	8	7.27	7.36	0.09	0.63	4769.87
Swim	1	66.71	65.98	0.73	2.88	8831.84
	2	70.15	69.18	0.97	3.55	9030.47
	4	63.95	63.46	0.49	3.79	9353.51
	8	37.99	37.15	0.84	5.93	9401.60
Applu	1	6.88	5.89	0.99	129.3	16784.15
	2	5.42	4.45	0.97	129.7	17646.31
	4	5.20	4.23	0.97	129.8	18848.32
	8	4.16	4.18	0.92	129.6	21104.30

respectively. These numbers are in sharp contrast with the respective total simulation times consumed: 55 hours, 110 hours and 230 hours! In addition to being efficient, our model is accurate (in terms of its prediction errors) and consistent (in terms of the trend exhibited by the errors). To understand these points further, Table 5 gives the absolute errors (in numbers) and the times to evaluate the three programs for (8KB, 64B). For the programs of the scale such as Applu, *EstimateMisses* yields the close to actual miss ratio in about 130 seconds for each associativity while the cache simulation runs for about 5 hours. This translates into a three orders of magnitude speedup over the cache simulator used. In terms of memory requirement, our model requires about 1.8MB, 7.1MB and 60.3MB to analyze

Tomcatv, Swim and Applu for the cache configuration (8KB, 64B), respectively.

7 Conclusion

To the best of our knowledge, this is the first work that demonstrates the feasibility of analyzing statically the cache behavior of whole programs with regular and compile-time predictable memory accesses. We have described, implemented and validated an analytical model for numerical codes, where the bulk of computations are expressed in loop nests operating on arrays. Our experimental results using kernels and complete programs indicate accurate cache miss estimates in substantially shorter amount of time than simulation. Our model can obtain predictions of miss ratios for program regions ranging from a single reference to the entire program. The causes for cache misses can be easily recovered from the specification (27). We believe the proposed mode is fast and accurate enough to guide compiler cache optimizations, in particular, those across nests (which are less-well developed [39]). We plan to apply our model for inter-nest cache optimizations in the future.

While this work represents a useful step towards an automatic analysis of whole programs, data-dependent constructs such as variable bounds, data-dependent IF conditionals and indirection arrays are still not statically analyzable. While Tomcatv, Swim and Applu are analyzable as a whole, the rest of the programs in SPECfp95 and Perfect Benchmarks are not despite that most of the lines can now be analyzed by our model. We plan to investigate techniques for their analysis. To go beyond FORTRAN 77, we need to cope with pointers and recursive calls. Some recent work [8], [12] on characterizing the reuse from some form of address traces is promising toward analytical modeling for data-dependent constructs.

There are many other benefits to static analytical modeling. An analytical model can be used as part of a general-purpose performance evaluation tool [34]. In addition, the mathematical formulas developed for characterizing cache misses may be potentially exploited to tighten the bound of the WCET (Worst-Case Execution Time) of a program [17], [55].

8 Acknowledgements

We would like to thank all the referees for their helpful comments and suggestions. This work has been supported by an Australian Research Council Grant A10007149.

References

[1] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *10th European Conference on Object-Oriented Programming (ECOOP'96)*, pages 142–166, 1996.

[2] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 85–96, 1997.

[3] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero. Ictineo: a tool for research on ILP. In *Supercomputing '96*, 1996. Research Exhibit "Polaris at Work".

[4] D. F. Bacon, J.-H. Chow, D.-C. R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *CASCON'94*, pages 270–282, Nov. 1994.

[5] N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '00)*, 2000.

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software — Practice and Experience*, 25:249–369, 1992.

[7] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI '01)*, pages 286–297, 2001.

[8] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 191–202, 2001.

[9] P. Clauss. Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In *ACM International Conference on Supercomputing (ICS'96)*, pages 278–285, Philadelphia, 1996.

[10] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, Jun. 1995.

[11] M. DeGroot. *Probability and statistics*. Addison-Wesley, 1998.

[12] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, 2000.

[13] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *2001 International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, Apr. 2001.

[14] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[15] P. Feautrier. Parametric integer programming. *Operations Research*, 22:243–268, 1988.

[16] P. Feautrier. Automatic parallelization in the polytope model. In G. R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer Verlag, 1996.

[17] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time System (LCTRTS'97)*,

pages 37–46, 1997.

[18] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *4th Workshop on Compilers for Parallel Computers*, pages 328–343, 1991.

[19] B. B. Fraguela, R. Doallo, and E. L. Zapata. Modeling set associative caches behavior for irregular computations. *ACM Performance Evaluation Review*, 26(1):192–201, June 1998.

[20] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

[21] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[22] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[23] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):1028–1075, 1999.

[24] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *5th Workshop on Languages, Compilers and Run-time Systems for scalable computers*, Rochester, May 2000.

[25] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative caches. *IEEE Transactions on Computers*, 48(10):1009–1024, Oct. 1999.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1st edition, 1996.

[27] M. Hill. DineroIII: a uniprocessor cache simulator. http://www.cs.wisc.edu/~larus/warts.html.

[28] S. C. V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *ACM International Conference on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, Jun. 1999.

[29] M. Kandemir, A. Choudhary, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb. 1999.

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

[31] I. Kodukul, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI'97)*, pages 346–357, Las Vegas,, 1997.

[32] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, pages 63–74, Santa Clara, Calif., Apr. 1991.

[33] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer Verlag, 1993.

[34] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *ACM SIGMETRICS'92 Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[35] The PIP System. Solving systems of affine (in)equalities: PIP's user's guide.

http://www.prism.uvsq.fr/~paf.

[36] The SUIF Compiler Group. SUIF: An infrastructure for research on parallelizing and optimizing compilers. http://suif.stanford.edu.

[37] S. McFarling. Program optimization for instruction caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 183–191, Apr. 1989.

[38] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, Jul. 1996.

[39] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Sept. 1999.

[40] J. M. Mellor-Crummey, D. B. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[41] A. K. Porterfield. *Software Methods for improvement of cache performance on supercomputer applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[42] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, 35(8):102–114, Aug. 1992.

[43] W. Pugh. Counting solutions to Presburger formulas: how and why. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 121–134, 1994.

[44] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, 1998.

[45] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.

[46] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a Java just-in-time compiler. In *2nd Java Virtual Machine Research and Technology Symposium*, San Francisco, 2002.

[47] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *ACM SIGMETRICS'94 Conference on Measurement and Modeling of Computer Systems*, pages 261–271, 1994.

[48] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for accessing when data copying should be used to eliminate cache conflicts. In *Supercomputing'93*, pages 410–419, 1993.

[49] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing'92*, pages 578–587, 1992.

[50] J. Torrellas, C. Xia, and R. L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.

[51] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(3):128–170, Sept. 1997.

[52] X. Vera, J. Llosa, and A. González. Near-optimal padding for removing conflict misses. In *15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02)*, Maryland, July 2002. Springer Verlag.

[53] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *European Conference on Parallel Computing (Europar'00)*, 2000.

[54] X. Vera and J. Xue. Let's study whole-program cache behaviour analytically. In *International Sympo-*

sium on High Performance Computer Architecture (HPCA-8), pages 175–186, Cambridge, Feb. 2002.

[55] R. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis of data caches and set-associative caches. In *3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Montreal, Jun. 1997.

[56] D. Wilde. A library for doing polyhedral operations. Technical Report 785, Oregon State University, 1993.

[57] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Toronto, Ont., Jun. 1991.

[58] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[59] J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Boston, Aug. 2000.

[60] J. Xue and C.-H. Huang. Reuse-driven tiling for improving data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.

Appendix

A Reducing the Evaluation Time of MRPA_R

Given $S_{R,R'}$ constructed in *FindMRPA*, we define the following set:

$$S_R \;=\; \bigcup_{R' \in RefSet} S_{R,R'} \tag{28}$$

Let us introduce the notation $\mathcal{R}(\vec{a})$ just in this appendix to represent the reference of which $\vec{a}$ is an access. The function MRPA_R can be expressed equivalently in terms of S_R as follows:

$$\mathsf{MRPA}_R(\vec{a}) = \max_{\prec}\{\vec{b} \in S_R \mid \vec{b} \in \mathcal{M}_{\mathcal{R}(\vec{b})}, ml_R(\vec{a}) = ml_{\mathcal{R}(\vec{b})}(\vec{b})\} \tag{29}$$

We discuss two strategies for reducing the time on spent evaluating MRPA_R for a given access $\vec{a}$ during behavior analysis. One is to remove some redundant elements from $S_{R,R'}$ that may be introduced in the two analytical cases of *FindMRPA*. The other is to sort S_R so that the impact of the remaining redundant elements is marginalized. An example is given to show that the latter is significantly more critical than the former.

In the first analytical case, the function $\mathsf{MRPA}_{R,R'}$ found is correct for any $\mathcal{M}_R$ and $\mathcal{M}_{R'}$. In the second case, $\mathsf{MRPA}_{R,R'}$ works for any $\mathcal{M}_R$ but $\mathcal{M}_{R'}$ must be rectangular. In numerical codes, $\mathcal{M}_R$ and $\mathcal{M}_{R'}$ are simple. By exploiting their geometrical relationship, a number of simplifications are possible. Let us mention two general rules, which have been specialized for common cases.

An element $\vec{b} \in S_{R,R'}$ is *redundant* if the $\mathsf{MRPA}_{R,R'}$ stays unchanged when $\vec{b}$ is removed.

- **Rule-A.** Let $\vec{b} \in S_{R,R'}$. Then $\vec{b}$ is redundant if $\forall\ \vec{a} \in \mathcal{M}_R : \vec{b} \notin \mathcal{M}_{R'}$.
- **Rule-B.** Let $\vec{b}, \vec{b}' \in S_{R,R'}$ such that $\vec{b}' \prec \vec{b}$. Then $\vec{b}'$ is redundant w.r.t $\vec{b}$ if (a) $\forall\ \vec{a} \in \mathcal{M}_R$: $\vec{b}' \in \mathcal{M}_{R'} \Rightarrow \vec{b} \in \mathcal{M}_{R'}$ and (b) $\forall\ \vec{a} \in \mathcal{M}_R : ml_R(\vec{a}) = ml_{R'}(\vec{b}') \Rightarrow ml_R(\vec{a}) = ml_{R'}(\vec{b})$.

In *FindMRPA*, each element $\vec{b}^*$ added to $S_{R,R'}$ in lines 22 and 38 has the form $\vec{b}^*_I = (i_1 - d_1, \dots, i_n - d_n)$. Thus, $\vec{a} - \vec{b}^*$ is an integer vector. By sorting S_R according to $\succ$, we can significantly reduce the average number of comparisons required for evaluating MRPA_R for the accesses of R. As a result, the impact of any redundant elements in S_R is reduced.

Theorem 7: *Consider* $\mathsf{MRPA}_R(\vec{a})$ *given in (29), where* $\vec{a}_I = (i_1, \dots, i_n)$. *Assume that every* $\vec{b} \in S_R$ *has the form* $\vec{b}_I = (i_1 - d_1, \dots, i_n - d_n)$, *where* $(d_1, \dots, d_n) \in \mathbb{Z}^n$. *Let all* r *elements of* S_R *be sorted as* $\vec{b}_1, \dots, \vec{b}_r$ *such that* $(\vec{a} - \vec{b}_k) \prec (\vec{a} - \vec{b}_{k+1})$. *Let* $\vec{b}_f$ *be the first in the list such that* $\vec{b}_f \in \mathcal{M}_{\mathcal{R}(\vec{b}_f)}$ *and* $ml_R(\vec{a}) = ml_{\mathcal{R}(\vec{b}_f)}(\vec{b}_f)$. *Then* $\mathsf{MRPA}_R(\vec{a}) = \vec{b}_f$.

Proof: The definition of $\max_{\prec}$ and that of $\mathsf{MRPA}_{R,R'}$ in (29). ■

As Example 3 shows, $\vec{b}^*$ found in the last case of *FindMRPA* may not have the desired form as in Theorem 7. A version of this theorem can then be generalized in an obvious way.

Example 7: *Consider the four subsets S_1 to S_4 that make up $S_{X(k,i),X(k,i)}$ in Example 5. S_2 can be ignored. This becomes clear if we set $\vec{b'} = (1, i_1, 1, i_2 - 1, 1, i_3 - \ell, 2)$ from S_2 and $\vec{b} = (1, i_1, 1, i_2, 1, i_3 - \ell, 2)$ from S_3 and then apply Rule-B. By further applying Rule-B to the elements in S_3, we find that $(1, i_1, 1, i_2, 1, i_3 - 3, 2)$ and $(1, i_1, 1, i_2, 1, i_3 - 2, 2)$ are both redundant w.r.t $(1, i_1, 1, i_2, 1, i_3 - 1, 2)$. Let $S'_3 = \{(1, i_1, 1, i_2, 1, i_3 - 1, 2)\}$. Let us consider S_4. Since $1 \leqslant i_3 \leqslant N$ and $1 \leqslant i_3 + N - \ell \leqslant N$, then $\ell \geqslant 1$ holds. By Rule-A, all elements in S_4 such that $\ell < 1$ are redundant. Among the three elements left in S_4, $(1, i_1 - 1, 1, N, 1, i_3 + N - 3, 2)$ and $(1, i_1 - 1, 1, N, 1, i_3 + N - 2, 2)$ are both redundant w.r.t $(1, i_1 - 1, 1, N, 1, i_3 + N - 1, 2)$ by Rule-B. Let $S'_4 = \{(1, i_1 - 1, 1, N, 1, i_3 + N - 1, 2)\}$. Thus, we have reduced $S_{X(k,i),X(k,i)}$ given in Example 5 to its subset: $S_{X(k,i),X(k,i)} = S_1 \cup S'_3 \cup S'_4$.*

$X(k, i)$ is the only reference to X. Thus, $S_{X(k,i)} = S_{X(k,i),X(k,i)}$ and $\mathsf{MRPA}_{X(k,i)} = \mathsf{MRPA}_{X(k,i),X(k,i)}$. Note that $|S_{X(k,i)}| = 6$. By Theorem 7, we sort $S_{X(k,i)}$ into a list, which is assumed to contain $\perp$ as its last, i.e., 7th element. When $N = 100$, $X(k, i)$ generates $N^3 = 10^6$ accesses. Let C_k be the number of accesses with the k-th element on the list as its MRPA. We have $C_1 = 740,000$, $C_2 = 237,600$, $C_3 = 9,900$, $C_4 = 0$, $C_5 = 9,900$, $C_6 = 99$ and $C_7 = 2501$. Thus, the average number of comparisons required for evaluating $\mathsf{MRPA}_{X(k,i)}$ for all accesses of $X(k, i)$ is found to be 1.31. If we did not eliminate any redundant elements from $S_{X(k,i)}$, i.e., $S_{X(k,i),X(k,i)}$, we would have $|S_{X(k,i)}| = 17$. The average number of comparisons would increase only to 1.85. Thus, the impact of Rule-B is more significant than that of Rule-A in reducing analysis time.

B The kernel codes

```
PROGRAM Hydro_K
REAL*8 ZA, ZP, ZQ, ZR, ZM, ZB, ZU, ZV, ZZ
DIMENSION ZA(JN+1,KN+1), ZP(JN+1,KN+1), ZQ(JN+1,KN+1), ZR(JN+1,KN+1), ZM(JN+1,KN+1))
DIMENSION ZB(JN+1,KN+1), ZU(JN+1,KN+1), ZV(JN+1,KN+1), ZZ(JN+1,KN+1)
T= 0.003700D0
S=0.004100D0
DO k= 2,KN
  DO j= 2,JN
    ZA(j,k)=(ZP(j-1,k+1)+ZQ(j-1,k+1)-ZP(j-1,k)-ZQ(j-1,k))*(ZR(j,k)+ZR(j-1,k))/(ZM(j-1,k)+ZM(j-1,k+1))
    ZB(j,k)= (ZP(j-1,k)+ZQ(j-1,k)-ZP(j,k)-ZQ(j,k))*(ZR(j,k)+ZR(j,k-1))/(ZM(j,k)+ZM(j-1,k))
  ENDDO
ENDDO
DO k= 2,KN
  DO j= 2,JN
    ZU(j,k)= ZU(j,k)+S*(ZA(j,k)*(ZZ(j,k)-ZZ(j+1,k))-ZA(j-1,k)*(ZZ(j,k)-ZZ(j-1,k))
               -ZB(j,k)*(ZZ(j,k)-ZZ(j,k-1))+ZB(j,k+1) *(ZZ(j,k)-ZZ(j,k+1)))
    ZV(j,k)= ZV(j,k)+S*(ZA(j,k)*(ZR(j,k)-ZR(j+1,k))-ZA(j-1,k) *(ZR(j,k)-ZR(j-1,k))
               -ZB(j,k) *(ZR(j,k)-ZR(j,k-1))+ZB(j,k+1) *(ZR(j,k)-ZR(j,k+1)))
  ENDDO
ENDDO
DO k= 2,KN
  DO j= 2,JN
    ZR(j,k)= ZR(j,k)+T*ZU(j,k)
    ZZ(j,k)= ZZ(j,k)+T*ZV(j,k)
  ENDDO
ENDDO
END
```

```
PROGRAM MGRID_K
REAL*8 U,Z
DIMENSION U(M,M,M), Z(M,M,M)
DO 400 I3=2,M-1
  DO 200 I2=2,M-1
    DO 100 I1=2,M-1
      U(2*I1-1,2*I2-1,2*I3-1)=U(2*I1-1,2*I2-1,2*I3-1)
        +Z(I1,I2,I3)
100 CONTINUE
    DO 200 I1=2,M-1
      U(2*I1-2,2*I2-1,2*I3-1)=U(2*I1-2,2*I2-1,2*I3-1)
        +0.5D0*(Z(I1-1,I2,I3)+Z(I1,I2,I3))
200 CONTINUE
  DO 400 I2=2,M-1
    DO 300 I1=2,M-1
      U(2*I1-1,2*I2-2,2*I3-1)=U(2*I1-1,2*I2-2,2*I3-1)
        +0.5D0*(Z(I1,I2-1,I3)+Z(I1,I2,I3))
300 CONTINUE
    DO 400 I1=2,M-1
      U(2*I1-2,2*I2-2,2*I3-1)=U(2*I1-2,2*I2-2,2*I3-1)
        +0.25D0*(Z(I1-1,I2-1,I3)+Z(I1-1,I2,I3)
        +Z(I1, I2-1,I3)+Z(I1, I2,I3))
400 CONTINUE
END
```

```
PROGRAM MMT
REAL*8 A, B, D, WB
DIMENSION A(N,N), B(N,N), D(N,N), WB(N.N)
DO J2 = 1,N,BJ
  DO K2 = 1,N,BK
    DO J=J2,J2+BJ-1
      DO K=K2,K2+BK-1
        WB(J-J2+1,K-K2+1)=B(K,J)
      ENDDO
    ENDDO
    DO I = 1,N
      DO K=K2,K2+BK-1
        RA=A(I,K)
        DO J=J2,J2+BJ-1
          D(I,J)=D(I,J)+
               WB(J-J2+1,K-K2+1)*RA
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
END
```

```
PROGRAM MM
REAL*8 X(N,N), Y(N,N), Z(N,N)
DO i = 1,N
  DO j = 1,N
    Z(j,i) = 0.0
    DO k = 1,N
      Z(j,i)=Z(j,i)+X(k,i)*Y(j,k)
    ENDDO
  ENDDO
ENDDO
END
```

```
PROGRAM LWSI_K
DOUBLE PRECISION xt, yt, xc, yc, zc
DOUBLE PRECISION zero, wsin, wcos, z, xs
DIMENSION xc(natoms, ns), yc(natoms, ns)
DIMENSION zc (natoms, ns), xt (natoms)
DIMENSION wsin(1), wcos(1), zero(1), z(1)
DIMENSION xs(1), yt (natoms)
DO i = 1, ns, 1
  xt(1) = xt(2)+wcos(1)
  xt(3) = xt(1)
  yt(2) = zero(1)
  DO j = 1, ns, 1
    yt(1) = yt(2)+wsin(1)
    yt(3) = yt(2)-wsin(1)
    z(1) = zero(1)
    DO k = 1, ns, 1
      DO l = 1, natoms, 1
      xc(l,k) = xt(l)
      yc(l,k) = yt(l)
      zc(l,k) = z(1)
      ENDDO
      z(1) = z(1)+xs(1)
    ENDDO
    yt(2) = yt(2)+xs(1)
  ENDDO
  xt(2) = xt(2)+xs(1)
ENDDO
END
```