Systematic Analysis of the Cache Behavior of Irregular Codes

Diego Andrade Canosa

Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems
University of A Coruña, Spain

PhD THESIS

Systematic Analysis of the Cache Behavior of Irregular Codes

Diego Andrade Canosa

March of 2007

PhD Advisors:
Basilio B. Fraguela Rodríguez
and Ramón Doallo Biempica

Dr. Basilio B. Fraguela Rodríguez
Profesor Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidad de A Coruña

Dr. Ramón Doallo Biempica
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidad de A Coruña

CERTIFICAN

Que la memoria titulada "*Systematic Analysis of the Cache Behavior of Irregular Codes*" ha sido realizada por D. Diego Andrade Canosa bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidad de A Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Informática.

A Coruña, 20 de Diciembre del 2006

Fdo.: Basilio B. Fraguela Rodríguez
Codirector de la Tesis Doctoral

Fdo.: Ramón Doallo Biempica
Codirector de la Tesis Doctoral

Fdo.: Luís Castedo Ribas
Director del Dpto. de Electrónica y Sistemas

Resumen de la tesis

Introducción

Existe una enorme diferencia entre la velocidad del procesador y la de la memoria. Esta diferencia convierte a la memoria en un cuello de botella que limita el rendimiento de los computadores. La jerarquía de memoria se utiliza para tratar de atenuar en lo posible el efecto de este cuello de botella. Se compone de varios niveles formados cada uno de ellos por memorias implementadas usando diferentes tecnologías. Las memorias de los niveles superiores son muy rápidas, con velocidades próximas a las del procesador, pero su tamaño es pequeño. A medida que descendemos en la jerarquía las memorias se van haciendo más lentas pero pueden albergar una mayor cantidad de datos.

La memoria del nivel más bajo de la jerarquía del computador contiene toda la información disponible. A medida que ascendemos en la jerarquía, cada nivel contiene un subconjunto de la información contenida en el nivel inferior. El funcionamiento de las jerarquías de memoria es sencillo: cuando el procesador necesita un dato solicita al nivel superior de la jerarquía el bloque de memoria en el que está contenido. Si el bloque se encuentra en ese nivel la petición es satisfecha y se produce un acierto, mientras que si el dato no está disponible en ese nivel se produce un fallo y la petición es trasladada al nivel inferior. Esta petición se propaga hacia abajo en la jerarquía hasta que el dato se encuentra en alguno de los niveles. En el peor de los casos, la petición será satisfecha en el nivel más bajo de la jerarquía.

La jerarquía de memoria explota el principio de localidad que en mayor o menor medida cumplen la mayoría de los procesos ejecutados en un computador. Existen dos tipos de localidad:

- Localidad espacial: si un dato ha sido accedido en un momento dado existe

una alta probabilidad de que datos cercanos se accedan proximamente.

- Localidad temporal: si un dato ha sido accedido en un momento dado existe una alta probabilidad de que ese mismo dato vuelva a ser accedido próximamente.

Las jerarquías de memoria están diseñadas de tal forma que los bloques de memoria más recientemente accedidos van a estar albergados en los niveles superiores de la jerarquía. Está claro pues que su uso favorece la mejora del rendimiento del computador porque un alto porcentaje de los accesos a memoria serán resueltos en los niveles superiores de la jerarquía.

Empezando por el nivel superior, la jerarquía de memoria de un computador está compuesta típicamente por: los registros del computador, la memoría caché, dividida a su vez en varios niveles diferentes, la memoria principal y finalmente el nivel de almacenamiento secundario. Una mejora en la localidad del código a ejecutar mejoraría el rendimiento de la jerarquía de memoria y en consecuencia la del computador.

Existen muchas técnicas que tratan de mejorar la localidad de los códigos a ejecutar en un computador mejorando así el rendimiento de la memoria. Los diferentes niveles de caché son la parte de la jerarquía de memoria más usada por el procesador después de los registros. Por lo tanto es importante tener técnicas que nos permitan conocer de forma rápida y precisa el comportamiento de las cachés durante la ejecución de un código en un determinado computador. Estas técnicas pueden ser usadas por ejemplo para guiar procesos de optimización de cara a incrementar la localidad en los accesos de los programas. Dada la gran disparidad entre la velocidad de acceso a los datos en las caché en la memoria principal esto puede dar lugar a grandes reducciones en el tiempo de ejecución.

Las principales técnicas que se usan para estudiar el rendimiento de la memoría caché son:

- Simulación dirigida por traza: Se usa una traza de las direcciones de memoria accedidas durante la ejecución de un código determinado para medir mediante un simulador el comportamiento de la caché durante su ejecución. Los principales inconvenientes de esta técnica son que es necesario ejecutar el código para obtener la traza y que la simulación a menudo lleva más tiempo que la ejecución del código real. A cambio, obtenemos buenos niveles de exactitud en la medición del rendimiento.

- Contadores hardware: Los contadores hardware existen en algunas arquitecturas y miden una gran cantidad de eventos relacionados con el hardware, entre ellos muchos eventos relacionados con el comportamiento de la caché. Podemos usar estos contadores durante la ejecución del código para estudiar el comportamiento de la caché. El principal inconveniente es que estos contadores están presentes sólo en ciertas arquitecturas y que sigue siendo necesario ejecutar el código para medir el comportamiento de la caché. Como en el caso anterior, la precisión de las mediciones obtenidas es alta.

- Modelado analítico: Podemos utilizar un modelo analítico de la caché para obtener una predicción de su comportamiento. Como información de entrada se puede usar una traza de las direcciones de memoria accedidas por el programa o el propio código fuente a ejecutar. El tiempo necesario para obtener la predicción es menor que en las dos anteriores técnicas, pero en general suele tener menor precisión en sus predicciones y la clase de códigos que podemos modelar debe tener unas características determinadas.

El modelo analítico de las PME (*Probabilistic Miss Equations*) [31] usa como información de entrada el código fuente a ejecutar para obtener una estimación rápida y fiable del comportamiento de la memoría caché de un computador. El modelo PME está limitado a códigos en los que los patrones de acceso a las estructuras son regulares. Se han propuesto algunos modelados analíticos para códigos irregulares concretos basándose en las ideas del modelo PME [30], pero no existe una estrategia automatizable que aborde el modelado del comportamiento de la caché para esta clase de códigos. Nuestro propósito es crear una extensión al modelo PME que nos permita abordar de forma automática el modelado de códigos en los que los accesos a algunas estructuras de datos siguen patrones de acceso irregulares.

Metodología de Trabajo

Abordar el modelado de códigos irregulares utilizando como primera referencia un código de complejidad excesiva habría sido un enfoque erróneo del problema. La lógica impone realizar primero el modelado de un código sencillo e ir refinando sucesivamente el modelado sobre códigos de complejidad creciente.

Cuando se considera el primer código se propone una estrategia automatizable de modelado que trate de cubrir toda la complejidad de la clase de códigos a modelar.

Se deriva a mano el modelado para ese código utilizando la estrategia propuesta. Se compara la predicción del modelo propuesto con los resultados obtenidos por una simulación dirigida por traza considerando distintas configuraciones de la caché y distintos tamaños de las estructuras involucradas en el código. Lo más probable es que la primera aproximación no funcione bien en todos los casos. En este caso se trata de identificar las posibles causas de la divergencia entre el simulador y el modelo y se proponen modificaciones que mejoren la predicción. Una vez se consiga que la predicción del modelo sea fiable para un amplio rango de configuraciones caché y tamaños de las estructuras involucradas, consideraremos que nuestro modelo realiza bien el modelado de este código concreto.

Sin embargo, el objetivo de nuestro modelo es cubrir el modelado de cualquier código irregular, por lo tanto se elige un código un poco más complejo y se deriva el modelado a mano repitiendo el mismo proceso que en el caso anterior hasta que la predicción sea fiable. Es necesario comprobar que cualquier modificación del modelo no afecta a la fiabilidad en la predicción para códigos anteriores. Cuando se hayan modelado con éxito un número razonable de códigos de complejidad creciente consideraremos conseguido el objetivo de tener una estrategia general y automatizable para el modelado de códigos irregulares.

Contribuciones

La existencia de una referencia con un patrón de acceso irregular se puede deber a diversos motivos: referencias dependientes de una o varias sentencias condicionales, estructuras indexadas a través de los valores contenidos en otras estructuras de datos, la existencia de punteros en el código etc...

En este trabajo hemos considerado dos fuentes principales de irregularidad: la existencia de estructuras de datos afectadas por condicionales y los accesos a través de indirecciones donde una estructura es indexada utilizando los valores contenidos en otra estructura diferente. Se han propuesto extensiones automatizables del modelo PME para ambos casos.

En el caso de sentencias condicionales hemos propuesto una extensión [7, 8, 11, 9, 12] capaz de modelar referencias contenidas dentro de una o varias sentencias condicionales anidadas cuya verificación se determina dinámicamente, esto es, en tiempo de ejecución. Un ejemplo de sentencia de este tipo sería una en la que el valor de verdad de la sentencia condicional dependiese a su vez de una expresión en

la que apareciese involucrada una referencia a un array cuyo valor solo puede ser conocido en tiempo de ejecución. Se impone la restricción de que la probabilidad de que se verifique la condición de la sentencia condicional tiene que ser uniforme, es decir, tiene que ser siempre la misma cada vez que es evaluada, y su valor debe ser suministrado como un parámetro al modelo. En el caso de tener varias setencias condicionales, éstas deben de ser totalmente independientes entre si, es decir, el valor de verdad de una de ellas no depende del valor de verdad de la otra.

En los códigos con indirecciones hemos utilizado como referencia los códigos que realizan computaciones con matrices dispersas. Estas matrices son almacenadas utilizando diferentes formatos comprimidos cuya manipulación da lugar a la aparición de una gran cantidad de indirecciones en esta clase de códigos. Hemos propuesto una extensión automatizable del modelo PME [10, 14] para cubrir códigos con indirecciones en los que cada posición de la estructura de datos tiene una probabilidad uniforme de ser accedida a través de la indirección. En el caso de una matriz dispersa el que tenga una distribución uniforme supone que los valores no nulos de la misma están uniformemente distribuídos a lo largo de la matriz. Al igual que el caso de los condicionales esta probabilidad debe ser suministrada como un parámetro de entrada al modelo.

Examinando los conjuntos de datos de entrada típicamente manipulados por códigos que realizan computación con matrices dispersas, descubrimos que la mayoría de las matrices tienen sus valores no nulos concentrados únicamente sobre una banda limitada de las mismas. En un primer momento propusimos una nueva extensión que cubría el modelado de matrices banda asumiendo que los no nulos dentro de la banda estaban distribuídos uniformemente [14]. Sin embargo, la mayoría de las matrices banda no tienen los valores distribuidos de manera uniforme. Por ello hemos propuesto una nueva extensión del modelo [13] que permite el análisis preciso del comportamiento de la caché durante la manipulación de esta clase de conjuntos de entrada.

Hemos propuesto extensiones automatizables y modulares para el modelado de códigos irregulares tanto con sentencias condicionales como con indirecciones. La automatización efectiva del proceso requiere la extracción de la información utilizada por el modelo del código a ser analizado. En el caso de códigos irregulares esta información a menudo se encuentra enmascarada en el propio código y no contenida de forma explícita, por tanto necesitamos una herramienta de compilación que sea capaz de manejar información simbólica y hacer un análisis avanzado del código. En nuestro trabajo hemos utilizado el compilador XARK para automatizar el modelado

de códigos con indirecciones con una distribución uniforme [6, 13].

Conclusiones

La mayoría de los modelos analíticos de la caché existentes sólo cubren el modelado de códigos que tienen patrones de acceso regulares. El modelado de códigos irregulares o bien ha sido realizado ad-hoc para ciertos códigos o está basado en heurísticas que no obtienen buenos niveles de precisión. En este trabajo hemos propuesto extensiones que cubren las principales causas de aparición de irregularidad en los accesos de un código. El manejo de información estadística sobre los conjuntos de entrada por parte del modelo se ha mostrado como la clave para poder obtener buenas estimaciones sobre el comportamiento de esta clase de códigos sin necesidad de ejecutarlos.

Por una parte hemos propuesto una extensión automatizable y modular para códigos con sentencias condicionales donde la probabilidad de que cada condición sea cierta se mantiene uniforme en cada evaluación de la misma. La fiabilidad de esta extensión se ha verificado comparando las predicciones del modelo con los resultados de simulación dirigida por traza aplicando el modelo a mano sobre códigos de este tipo de creciente complejidad. La predicción mediante un modelo analítico del rendimiento de la caché durante la ejecución de un código dado es una tarea en si complicada. Considerar patrones de acceso irregulares aumenta considerablemente el grado de dificultad de realizar dicha predicción. A pesar de ellos los niveles de fiabilidad del modelo para códigos irregulares es alta. Además, a pesar de la mayor complejidad computacional de las nuevas fórmulas con respecto a las derivadas para códigos regulares, el tiempo de ejecución del modelo se mantiene extremadamente bajo, en concreto éste está siempre por debajo de un segundo para cada ejecución.

Hemos realizado otra extensión modular y automatizable para códigos con indirecciones. Los códigos que hemos tomado como ejemplo para realizar el modelado realizan computación con matrices dispersas. En una primera extensión hemos considerado solamente matrices dispersas uniformes, es decir, en las que los valores no nulos de la matriz se encuentran uniformemente esparcidos a lo largo de la misma. El modelo propuesto para este caso es capaz de obtener una predicción fiable del rendimiento de la caché en un muy corto periodo de tiempo. En un siguiente paso hemos estudiados colecciones como la la Harwell Boeing [28] y la NEP [18] que contienen un gran número de matrices típicas utilizadas en computación dispersa.

Hemos observado que un alto porcentaje de las matrices que contienen son banda, es decir, la mayoría de los valores no nulos están esparcidos a lo largo de una banda limitada de la matriz. Ello nos ha llevado a proponer una aproximación para abordar el modelado de códigos que manipulan esta clase de matrices. En este sentido, hemos propuesto primero una pequeña modificación para modelar el comportamiento de matrices banda uniforme y posteriormente otra extensión del modelo PME para analizar códigos que manipulan matrices en la que los valores no nulos no están uniformemente distribuidos dentro de la banda. La validación de estos modelos se efectuó modelando códigos de creciente complejidad que utilizan matrices dispersas tanto sintéticas como reales y comparando luego las predicciones del modelo con los resultados de simulaciones dirigidas por traza. El tiempo necesario para la ejecución del modelo se mantiene por debajo de un segundo incluso para casos en los que la ejecución del código objeto del análisis se prolonga durante muchos minutos. La fiabilidad de la predicción obtenida continúa siendo muy alta.

Una vez realizadas estas extensiones del modelo PME nos propusimos hacer la automatización efectiva de alguna de ellas. Para ello utilizamos una herramienta avanzada de compilación, el compilador XARK, capaz de extraer de los códigos a analizar la información que el modelo precisa. Una vez automatizado todo el proceso, abordamos el modelado automático de los códigos anteriores y de nuevos códigos contenidos dentro de la librería SPARSKIT [43] especializada en el tratamiento de matrices dispersas. Los resultados obtenidos muestran que las predicciones siguen siendo las mismas que las obtenidas cuando habíamos aplicado manualmente las ecuaciones sobre los códigos. Una de las preguntas latentes durante la aplicación del modelado a mano era cuánto tiempo llevaría construir las ecuaciones del modelo automáticamente utilizando como entrada la información de un compilador. El tiempo necesario para extraer la información del código por parte del compilador y utilizar ésta para generar una predicción del rendimiento usando el modelo sigue siendo realmente bajo y en algunos casos se mantiene varios órdenes de magnitud por debajo del tiempo de ejecución del código analizado.

Una de las aplicaciones de esta clase de modelos es servir de guía en un proceso de optimización. En este sentido, realizamos un experimento que consistió en utilizar el modelo PME como guía a la hora de decidir cuál era el ordenamiento óptimo de los lazos en el producto entre una matriz dispersa y una matriz densa. Se hicieron pruebas utilizando tanto matrices sintéticas como reales de diferentes tamaños, densidades y considerando diversas arquitecturas reales con distintas configuraciones de la caché. La decisión adoptada guiándonos por el modelo siempre coincidió con la

adoptada usando como referencia el tiempo de ejecución del código en la máquina real. Incluso en caso de matrices no uniformes, aunque cuantitativamente la predicción del modelo era a veces inexacta, la decisión tomada utilizando el modelo como guía era siempre la correcta.

La ampliación del ámbito de aplicación del modelo PME al campo de los códigos irregulares supone un gran paso adelante en la utilización efectiva de los modelos analíticos como alternativa a las técnicas tradicionales de simulación dirigida por traza y contadores hardware. Los códigos irregulares adolecen de falta de localidad y por lo tanto pueden obtener un gran beneficio de la aplicación de técnicas de optimización que la mejoren. Además estas extensiones mantienen intactas todas las características deseables en una técnica para el estudio del rendimiento de la memoria: fiabilidad en la predicción, rapidez en su ejecución y la posibilidad de conocer los entresijos del funcionamiento del código en lugar de simplemente obtener una única cifra indicativa del rendimiento.

Trabajo Futuro

En un futuro planeamos abordar el modelado del comportamiento de la caché en arquitecturas multinúcleo ya que éstas empiezan a ser cada vez más frecuentes hoy en día. La complejidad y novedad en el análisis de las jerarquías de memoria de estas arquitecturas, reside en la existencia de varios procesadores que pueden compartir uno o varios niveles de la caché. Trataremos de modelar esta situación usando el modelo PME como base. La automatización efectiva del modelo puede mejorarse tanto para códigos con indirecciones y una distribución banda de la matriz, como para códigos con sentencias condicionales.

Debio a su exactitud, rapidez y amplio ámbito de aplicación, este modelo se ha convertido en una poderosa herramienta para predecir el comportamiento de la caché. Planeamos usar el modelo para guiar optimizaciones tanto sobre códigos regulares como irregulares, además de las ilustradas en esta tesis. Optimizaciones tales como, la selección óptima del tamaño de bloque en la aplicación de la técnica de blocking o métodos que nos permitan guiar la prebúsqueda de datos usando las predicciones del modelo. Sería interesante usar las capacidades del modelo en el campo de los sistemas embebidos y comprobar como sus predicciones pueden ser usadas en esta clase de sistemas para mejorar su rendimiento. Se ha desarrollado poco trabajo en este área del modelado del comportamiento de la caché de esta clase

de sistemas. Intentaremos derivar estimaciones del mínimo y el máximo número de fallos en código irregulares y usarlos en aplicaciones tales como el cálculo del WCET (Worst Case Execution Time), un problema abierto en los sistemas embebidos.

Publicaciones

- D. Andrade, B.B. Fraguela, and R. Doallo. Efficient and accurate analytical modeling of the cache behavior of complete scientific codes. In *IASTED Intl. Conf. on Applied Simulation and Modelling 2003*, pages 106–111, Marbella, September 2003.

- D. Andrade, B.B. Fraguela, and R. Doallo. Modelado de caches ante códigos con condicionales dependientes de datos. In *Actas de las XIV Jornadas de Paralelismo*, pages 281–286, Leganés, September 2003.

- D. Andrade, B. B. Fraguela, and R. Doallo. Cache behavior modeling of codes with data-dependent conditionals. In Springer-Verlag, editor, *In Proceedings of Workshop on Software and Compilers for Embedded Systems*, volume 2826 of *Lecture Notes in Computer Science*, pages 373–387, Vienna, Austria, September 2003.

- D. Andrade, B.B. Fraguela, and R. Doallo. Modeling the cache behavior of codes with arbitrary data-dependent conditional structures. In Springer-Verlag, editor, *In Proceedings of the Asia-Pacific Computer Systems Architecture Conference*, volume 3189 of *Lecture Notes in Computer Science*, pages 44–57, Beijing, China, September 2004.

- D. Andrade, B.B. Fraguela, and R. Doallo. Modelado analítico automático del comportamiento de la caché para códigos con indirecciones. In *Actas de las XVI Jornadas de Paralelismo*, pages 321–328, Granada, September 2005.

- D. Andrade, M. Arenaz, B. B. Fraguela, J. Touriño, and R. Doallo. Automated and accurate cache behavior analysis for codes with irregular access patterns. *In Proceedings of Workshop on Compilers for Parallel Computers*, pages 179–193, A Coruña, Spain, January 2006.

- D. Andrade, B. B. Fraguela, and R. Doallo. Analytical modeling of codes with arbitrary data-dependent conditional structures. *Journal of Systems Architecture*, 52(7):394–410, July 2006.
- D. Andrade, B. B. Fraguela, and R. Doallo. Cache behavior modeling for codes involving banded matrices. In *Proc. of the 19th Intl Workshop on Languages and Compilers for Parallel Computing*, New Orleans, November 2006.
- D. Andrade, B. B. Fraguela, and R. Doallo. Precise automatable analytical modeling of the cache behavior of codes with indirections. *ACM Transactions on Architecture and Code Optimization*, 2007. Accepted for publication.
- D. Andrade, M. Arenaz, B. B. Fraguela, J. Touriño, and R. Doallo. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concurrency and Computation: Practice and Experience*, 2007. Accepted for publication.

Abstract

The performance of memory hierarchies, in which caches play an essential role, is critical in nowadays general-purpose and embedded computing systems because of the growing memory bottleneck problem. Unfortunately, cache behavior is very unstable and difficult to predict. This is particularly true in the presence of irregular access patterns, which exhibit little locality. Such patterns are very common for example in applications in which some references are guarded by conditional statements or in which pointers or compressed sparse matrices give place to indirections. Nevertheless, cache behavior in the presence of irregular access patterns has not been widely studied. In this thesis we present separated extensions of a systematic analytical modeling technique based on PMEs (Probabilistic Miss Equations) that allows the automated analysis of the cache behavior for codes that include data-dependent conditional structures and codes with irregular access patterns due to indirections, respectively. The model generates very accurate predictions despite the irregularities and has very low computing requirements, being the first model that gathers these desirable characteristics that can analyze automatically this kind of codes. These properties enable this model to help drive compiler optimizations. The PME model extension for codes with indirections has been integrated in the XARK compiler, a research compiler oriented to automatic kernel recognition in scientific codes. We show how to exploit the powerful information-gathering capabilities provided by this compiler to allow automated modeling of loop-oriented scientific codes.

To my big family.

Acknowledgements

This thesis is not the result only of my own effort; there are many people involved in this work whose support and dedication I want to acknowledge. First, I want to acknowledge my PhD supervisors Basilio and Ramón for the confidence they placed on me, and the support they gave me along all these years. I cannot forget my colleagues in the Department of Electronics and Systems because they made easier my experience in the lab during all these years, specially to Manuel Arenaz with whom i worked in the integration of the XARK compiler with the PME model. Finally, I want to acknowledge my parents because they have always been there.

I gratefully thank to the following institutions for funding this work: *Department of Electronics and Systems of A Coruña* for the human and material support, *University of A Coruña* for financing my attendance at some conferences, and *Xunta de Galicia* and *Spanish Government* for the projects 1FD97-0118-C02-02, TIC2001-3694-C02-02, TIN2004-07797-C02-02 and PGIDIT03-TIC10502PR.

Diego Andrade

"Le diable se cache dans les détails", Swiss Proverb

Contents

List of Tables

List of Figures

Preface

The performance of memory hierarchies, in which caches play an essential role, is critical in nowadays computing systems because of the growing memory bottleneck problem. Unfortunately, cache behavior is very unstable and difficult to predict. We need techniques that allow us to study accurately the cache behavior with a low computational cost, so they can be used for example as a guide in iterative optimization processes. Hardware counters and trace-driven simulators have been traditionally used to study the cache behavior. These methods are very accurate but they have a very high computational cost and they provide us a summarized characterization of the cache performance but not any insights about the studied behavior.

Analytical models try to predict the cache behavior using information from a trace of the memory addresses accessed by the code or from the source code to execute. Most analytical models only cover the modeling of codes with regular access patterns. The PME (Probabilitic Miss Equations) model [31] is an analytical model that can provide very accurate predictions of the cache behavior automatically with a low computational cost using information extracted from the source code to execute. Although the ideas of the PME model had been used to model some irregular kernels [30], there was not a general automatic strategy to model this kind of codes. This was a very important limitation for the application of this technique because irregular codes are relatively common and they have very little locality. As a result, a big performance increase can be obtained by applying different optimization techniques that improve the locality based on the predictions of a model. The main interest of this work is to propose a modular, extensible and automatic strategy for the modeling of codes with irregular access patterns.

The existence of references with an irregular access pattern can be due to different causes: references guarded by one or several conditional statements, arrays indexed using the values contained in other arrays, pointers etc... In this work, we have

considered two main sources of irregularity: the existence of references guarded by conditional statements and the accesses across indirections where an array is indexed using the values contained in another array. Extensions to the PME model have been proposed for both situations.

In the case of conditional statements we have proposed an extension [7, 8, 11, 9, 12] that can model references contained inside one or several conditional statements whose verification is determined dinamically, that is, at runtime. One example of this situation is a conditional statement the fulfillment of whose condition depends on an expression which involves an array reference whose value can only be determined at runtime. The probability of fulfillment must be uniform, that is, it must be the same in each one of its evaluations and its value must be provided to the model as a parameter. If there are several conditional statements they must be independent, that is the probability of fulfillment of each condition does not depend on the fulfillment of any other condition.

In the case of codes with indirections, we have used as a reference the codes that perform computations with sparse matrices. These matrices are stored using different compressed storage methods whose manipulation gives place to a big number of indirections. We have proposed an automatable extension of the PME model [10, 14] to cover this kind of codes where each position of the data structure accessed across the indirections has the same probability of being accessed. As in the case of the codes with conditionals, this probability is an input parameter of the model. In the case of an sparse matrix this implies that the non-zero values must be uniformly distributed on the matrix.

When we explore the typical input data collections for codes that perform sparse computations, we discover that most of these matrices have the majority of their non-zeros concentrated in a limited band. In a first step we proposed an small modification of the PME model which considered that the values were uniformly spread along the band [14]. But, most of the banded matrices have their values non-uniformly spread along the band. So a new PME model extesion was proposed [13] to cover this situation.

We have proposed automatable and modular extensions for the modeling of irregular codes both due to indirections and conditional statements. The effective automation of this proccess requires the extraction of the input data used by the model from the analyzed code. In the case of irregular codes this information is often masked in the code, so we need an advanced compilation tool capable of managing

symbolic information and performing an advanced analysis of the studied code. In our work we have used the XARK compiler framework for the automation of the modeling of codes with indirections and an uniform distribution of the values [6, 13].

The results of all the stages of this work have been validated by comparing the model predictions with the result of trace-driven simulations. The results obtained in all the cases reflect that the model is very accurate and that its execution is completed in a short time. The model execution always takes less than one second and in some cases this time is several orders of magnitude shorter than the one necessary for the execution of the analyzed code.

Objectives and Organization of this Thesis

The scope of application of the Probabilistic Miss Equation (PME) [31] model was limited to codes with regular access pattern. This work extends its scope to codes with irregular access patterns. In order to simplify the modeling, separated extensions are proposed for codes with data-dependent conditional statements and with indirections.

The extension for codes with conditional statements allows the PME model to predict the cache behavior of references guarded by this kind of sentences. The references can be affected by one or more conditionals with any kind of nesting between them. The conditions must follow an uniform distribution, that is, they must have an uniform probability of being true in each evaluation and if there are several conditional statements they must be independent.

The extension for codes with indirections allows the PME model to consider references in which an array, called the base array, is referenced using the values obtained from another array, called the index array. More than one level of indirection can be modeled by this extension. The model has been developed considering an uniform distribution of the values generated by the indirection, that is, all the positions in the base array have the same probability of being accessed using the indirection. A latter extension in this thesis allows the modeling of indirections with uniform and non-uniform band distributions. This distribution is very common in sparse matrices in Compressed Row Storage(CRS) format [19]. The codes that manipulate this kind of matrices are the main source of benchmarks used to test this extension.

These extensions are fully automatable and modular. The extension for codes with indirections is integrated with the XARK compiler to analyze this kind of codes automatically. The XARK compiler extracts the information from the source code of the program to analyze and passes it to the PME model implementation.

This thesis is organized as follows: Chapter 1 is a brief introduction to the problem of the cache performance study. It contains information about the different techniques used for this purpose, their main advantages and disadvantages. It includes a survey of most of the existing analytical models and their main characteristics.

Chapter 2 contains a description of the original automatable PME model that only covered codes with regular access patterns. The different stages of the PME model are described in detail: the miss probability estimation and the PME equation construction. No validation is included in this chapter because it belongs to previous works and its accuracy has been already demonstrated [31].

Chapter 3 describes the PME model extension that covers irregular codes due to data-dependent conditional statements. In this chapter, the scope of application of the extended PME model is established. The miss probability estimation step is adapted to cover the new situation. There is a description of the new type of PME that models the references that are guarded by conditional statements. Finally, the accuracy of this extension is validated.

In Chapter 4 the PME model extension for irregular codes due to indirections is covered. As in Chapter 3, there are several adaptations that must be done in the scope of application and the miss probability estimation process. Besides, new types of equations are added to the model to cover the new situations. This extension is also validated using several typical kernels that exhibit this kind of access pattern.

Chapter 5 covers the automation of the PME model extension for codes with irregular access patterns due to indirections. The information requirements of the PME model are described, and the interface between the PME model and the XARK compiler is described [15]. The role of the XARK compiler is to extract the information from the source code and provide it to the PME model in the form established by the interface between them. A brief introduction to the XARK compiler is also contained in this chapter.

Finally, the extension of the PME model for irregular codes has given place to several publications in the area of the study of the cache performance behavior

and prediction. The extension for codes with conditional statements is explained in [7, 8, 11, 9, 12]. The extension for codes with indirections has been split in several contributions: some of them covered the modeling for uniform and banded uniform distributions [10, 14] while a different contribution covered the modeling of non-uniform banded distributions [13]. The automation of the PME model extension for codes with indirections using the XARK compiler was covered in [6, 5].

Chapter 1

An Introduction to Cache Modeling

The gap between processor and memory speed is increasing year by year. Current architectures use a hierarchy of levels of memory [34] in order to try to cushion this gap. This hierarchy has fast small memories in the top levels and bigger but slower memories in the lower levels. From top to bottom, a typical hierarchy would be composed by the the processor registers, one or several levels of cache memory, the main memory and the secondary storage.

Each level in the hierarchy is divided in blocks. When the processor needs a memory item, the block that contains it is searched in the top level of the hierarchy. If it is not found, the request proceeds to the next lower level. The request is propagated this way down the levels in the hierarchy until the data is found. Once the block is found, it is loaded in all the levels above the one where it was found. When a block is found in a memory level, that access is considered a hit, otherwise it is a miss. The miss rate of a level is the ratio of accesses that result in a miss.

Cache memory blocks are termed lines and they are organized in sets. All the sets have the same number of lines. This number is called the degree of associativity of the cache. When a memory block is loaded in the cache, it can be stored in exactly only one set, but any of the lines in the set can hold the line. Depending on the possible location of a memory block in the cache we distinguish three types of cache organizations:

- Direct mapped: Each set contains only one line, so each block can only be stored in exactly one line in the cache. This line is usually calculated as $address \bmod num$ where $address$ stands for the memory block address and

num stands for the number of lines in the cache.

- Fully associative: If the block can be stored in any cache line because the cache has only one set that contains all its lines.
- Set-associative: The cache is divided in sets of K cache lines each, where K is the degree of associativity. Each memory block can only be mapped to a specific set. The block can be loaded in any line inside that set. The set where a given block is stored is selected using the function $address \bmod N_c$, where *address* is the block address and N_c is the number of cache sets.

When a memory line is brought to the cache, it can be stored in any line of the cache (fully associative cache), only in a given line (direct mapped) or in a given set of lines (associative cache). If all the candidate lines to store a memory block contain valid information then, one of them must be selected to be replaced and make room for the new line. This selection is done according to a replacement policy. In our work we will use the most common replacement policy, the Less Recently Used (LRU) policy, in which the less recently referenced line is selected.

The three types of cache misses are:

- Cold or compulsory misses: Since data is brought to the cache on demand, the first access to a memory block results necessarily in a miss.
- Capacity miss: If the cache cannot store all the blocks accessed during the program execution, then, there are blocks that are replaced during the execution. Latter references to such blocks result in capacity misses.
- Interference misses: They happen in direct mapped and set-associative caches. In these kinds of caches a block can be replaced during the execution because many blocks are mapped to its cache line or set of lines even if there is enough space in the cache to hold all the data.

Memory hierarchies store the most recently used memory blocks in the top levels exploiting the locality typically found in the memory references of applications. Locality appears when the same data is accessed multiple times in a short period of time. There are two types of locality [45]:

- Temporal locality : when a single memory item is accessed multiple times in a short period of time.

- Spatial locality : when two close memory items belonging to the same block are accessed in a short period of time.

The memory performance can be improved by:

- Reducing the miss penalty: the miss penalty is the time required to solve an access that misses in a level of the hierarchy.
- Reducing the hit time: the hit time is the time necessary to access a data item when the corresponding block is found in a level of the memory hierarchy.
- Reducing the miss rate: what can be achieved by improving the locality of the code to execute.

The more the locality of the code is improved, the more data requests from the processor will be solved in some of the top levels of the memory hierarchy. In the last years a large number of techniques to study the cache behavior have appeared. These techniques can be used in optimization processes [55, 2, 3] for improving the locality of a given code or for choosing the optimal cache configuration.

1.1. Techniques to Study the Cache Behavior

Three techniques are used nowadays to study the cache behavior: trace-driven simulation, hardware counters and analytical modeling. Each technique is explained in turn.

1.1.1. Trace-driven Simulation

One of the first techniques proposed to study the cache behavior is trace-driven simulation [50, 57]. This technique consists in simulating the behavior of a given cache configuration for a sequence of memory accesses. This sequence of memory references of a given program is called address trace. The address trace is processed using a program that simulates the proposed cache configuration and outputs a description of the cache behavior for the considered accesses. Generally, accurate estimations can be obtained using this technique, but it presents some problems :

- Trace collection is not a trivial task in complex scenarios where several different processes can be running concurrently, including the operating system and where the code is dynamically linked or compiled. An usual way to perform the trace collection is by means of an instrumented version of the code whose behavior is to be analyzed.
- A reduction of the trace size is often necessary because the address trace is typically very large and it may need several gigabytes of storage space.
- Trace processing is a time consuming task and it usually requires much more time than the execution of the original code. Some approaches [38] try to reduce the number of instructions simulated by selecting a representative set of instructions while trying to avoid a loss of accuracy in the simulation. However, this still requires more time than the execution of the code to analyze.

Consequently, trace-driven memory simulation can generate accurate estimations at the expense of a high consumption of resources and it is thus inadequate to guide compiler optimizations.

1.1.2. Hardware Counters

Another technique suitable for studying the cache behavior is the use of hardware counters [4, 25, 58], which are available in most current architectures. Hardware counters are registers that can take account of information about a wide range of events during the execution of a given code. There are several registers which can track information about a big number of events related to the cache behavior and that can be used in its study. These registers can work in two different modes :

- In counting mode the registers are used for obtaining aggregate counts of occurrences of specific events.
- While in sampling mode the frequency of event occurrences in different scopes of the program can be tracked.

This information can provide a precise picture of what is happening in the cache memories. This information is extracted using an interface that is different in each architecture although in the last years standard interfaces such as PAPI [20] have been defined. However, hardware counters present a set of associated problems:

- Hardware counters are present only in some architectures, and although they are available in most modern architectures there is a wide variation between the registers available in different systems.
- The computational cost of this technique is high because it is necessary to execute the program to collect the information from the hardware counters.
- The usage of hardware counters introduces an overhead in the execution due to extra instructions and can cause cache pollution, thus changing the cache behavior of the monitored program.

1.1.3. Analytical Models

Analytical models try to obtain accurate estimations of the cache behavior with a lower computational cost than the two previous approaches. They try to construct an analytical model that can predict the cache behavior during the execution of a given code. There are analytical models that use as their input an address trace of the memory addresses accessed during the execution of the code, while other models use as input the source code to execute.

This technique provides a more detailed insight of the observed cache behavior. Its main drawback is its limited scope of application and the limited degree of accuracy of some models. Most analytical models only consider codes with regular access patterns, although the analysis of the behavior of codes with irregular access patterns is of great interest, as they exhibit less locality, and thus caches do not perform well for them.

Previous Works in Analytical Modeling

As we said before, there are mixed techniques based in analytical models that use information extracted from an address trace obtained in a previous execution of the code, like the technique proposed by Agarwal et al. [1]. This model can derive miss rates for different proposed cache organizations and workloads from the information provided in an address trace. The parameters used in this model are the probability of access to data or code lines and the probability of accessing consecutive positions. This work is mainly focused to a multiprogrammed system and pays a lot of attention to the influence of the operative system. For this purpose, it considers three different categories of misses: the cold misses that happen when the cache is

initially filled, the non-stationary misses due to a change in the program data set and the interference misses due to collisions in random positions in the data sets.

The model proposed by Quong in [41] uses also information extracted from a trace of the memory addresses accessed by the program. It also considers that each block has an uniform probability of being mapped to each cache set. This model considers the misses as a whole set and it is based in the calculation of the average region accessed between two consecutive accesses to the same line. However it estimates the number of misses produced in the instruction cache and not the data cache.

The work proposed by Buck and Singhal in [21] uses a trace for studying the behavior of a fully associative cache. It is based in the Independent Reference Model (IRM) proposed by King in [35], which assumes that the probability of accessing each line is constant along time. This assumption simplifies the model but it dramatically reduces its scope of application, because in the real world programs work only with a limited data set in each period of time.

In [26] Din and Zhong use information from very large address traces obtained during the execution of a program using a given input data for predicting its behavior in a future program run. The reuse distance between two consecutive accesses is measured using an optimized tree representation. It is only suitable for programs that have a consistent pattern that make it predictable along different program executions.

Another set of methods try to model the cache behavior using information from the source code to execute. Some techniques can perform the modeling of a given optimization technique applied on a specific code, like the one described by Simecek and Tvrdik in [44], that is centered in the application of the dynamical loop reversal optimization over the Cholesky factorization. This work considers any kind of cache configuration with Less Recently Used (LRU) block replacement policy. Its main advantage with respect to other analytical models is that it makes a detailed modeling of recursive calls not considered by other works in this field.

The work presented by Temam et al. in [47] is based in the ideas introduced in [48]. Its application is restricted only to direct-mapped caches and codes with regular access patterns. It considers cold, capacity and interference misses.

There are some proposals which try to cover the modeling of a wider scope of codes. Chatterjee et al. propose in [42] a detailed model based in Presburger

formulas that handles regular codes with either perfectly or non-perfectly nested loops giving accurate estimations. The main limitations of this model are its high computational cost and that it only supports modest levels of associativity in the cache configuration. A different approach was proposed by Harper et al. [33]. The estimation provided is not so accurate and it supports the modeling of perfectly or non-perfectly nested regular loop constructs for any kind of cache configuration. Cache miss equations (CMEs) are used by Ghosh et al. [32] for analyzing a set of perfectly nested regular loops considering cache configurations with any level of associativity; its support for non-perfectly nested loops is weak. The CMEs are a set of Diophantine equations that are obtained once for each considered code and the solutions are obtained for every different situation analyzed changing some of the variables by the corresponding values. CMEs are used also by Vera and Xue [56, 54] for analyzing perfectly nested regular loops. It has a better support for a significant subset of non-perfectly nested loops and statically analyzable conditional statements [53].

The Probabilistic Miss Equation (PME) model [31] is a probabilistic model of the source code capable of analyzing the cache behavior of scientific codes with both perfectly and non-perfectly nested loops. The model has a low computational cost and the prediction obtained is quite accurate, but its scope is limited to codes with regular access patterns.

As for the modeling of codes with irregular access patterns, the models found in the bibliography are not systematic enough to be automated or do not provide accurate predictions. The method proposed by Temam and Jalby in [49] studies the autointerferences in the vector involved in a sparse matrix vector product in a direct-mapped cache, but it does not consider the interference with the other data structures in the code. The approach described by Ladner et al. in [37] is an ad-hoc model whose scope of application is limited only to direct-mapped caches and it does not consider the interaction between different interleaved access patterns. These limitations were overcome in the probabilistic model [30] but it was not systematic enough to be automatable.

Some works have tried to approach the modeling of the cache behavior in codes with irregular access patterns automatically. The indirect accesses model [22, 24, 23] of Cascaval and Padua is integrated in a compiler framework, but it is a simple and inaccurate heuristic that estimates the number of cache lines accessed rather than the real number of misses. For example, it does not take into account the distribution of the irregular accesses and it does not account for conflict misses, since it assumes

Table 1.1: Main characteristics of the existing analytical models of the cache behavior

Analytical Model	Input	Associativity	Scope	Automatic	Accuracy
Agarwal [1]	Trace	Any	Both	Yes	Low
Quong [41]	Trace	Any	Both	Yes	Low
Buck and Singhal [21]	Trace	Fully	Both	Yes	High
Ding and Zhong [26]	Trace	Any	Regular	Yes	High
SPLAT [46]	Trace	Any	Both	Yes	Low
Temam et al. [47]	Source	Direct	Regular	No	High
Simececk and Tvrdik [44]	Source	Any	Ad hoc	No	High
Temam and Jalby [49]	Source	Direct	Ad-hoc	Yes	High
Chatterjee et al. [42]	Source	Low	Regular	No	High
Harper et a. [33]	Source	Any	Regular	No	Low
Ghosh et al. [32]	Source	Any	Regular	Yes	High
Vera and Xue [56]	Source	Any	Regular	Yes	High
Ladner et al. [37]	Source	Direct	Both	No	High
Cascaval and Padua [23]	Source	Any	Both	Yes	Low
PME [31]	Source	Any	Regular	Yes	High
Fraguela et al. [30]	Source	Any	Ad-hoc	No	High

a fully-associative cache. Another approach is that of SPLAT [46], a tool that analyzes codes in several phases: the reuse and volume phases, where compulsory and capacity misses are computed respectively considering a fully associative cache; and the interference phase, where conflict misses are calculated considering a direct-mapped cache. Irregular accesses due to conditional statements and loops with a variable number of iterations are modeled using the information derived from a previous profiling of the code.

Table 1.1 contains a summarized overview of the main characteristics of the analytical models we have studied in this section. The first column specifies whether the model uses as input the trace of the memory addresses or the source code, the second column if it can model the behavior of any cache [1], only direct mapped caches or only caches with a low degree of associativity. The third column describes the scope of application of the model. It classifies a model in one of three categories : those that can model the cache behavior of regular codes, irregular codes or both types of codes. The fourth column indicates whether that model has been automated, and the fifth one contains the degree of accuracy of the obtained prediction. The main drawback in trace-based methods [1, 41, 21, 26, 46] is their high com-

putational cost, because the real code must be executed to obtain the input trace. The source-based methods are more affected by the problem of the limited scope of application. They are mainly limited to regular codes [47, 44, 42, 33, 32]. Some of them cover irregular computation [37, 23] but either they are not automated or the accuracy of the provided prediction is low.

Chapter 2

The PME Model

2.1. Introduction

The PME (Probabilistic Miss Equations) model [31] estimates accurately the number of misses in a cache during the execution of a given code with a low computational cost. Cache misses can be classified into three groups. Compulsory or cold misses take place the first time a given memory line is accessed, since data lines are loaded in the cache on demand in this first access. Although a memory had been accessed previously, it may not be found in the cache in an attempt to reuse it. This can be due to the fact that the cache is not large enough to store all the data accessed by the studied code (capacity miss) or due to other data items having been mapped to the same cache set and evicting then that line from the cache (interference miss). In our work, we consider both capacity and interference misses together as interference misses because both kinds of misses happen when a line that had been referenced previously has been ejected from the cache since its last access due to interferences with other lines mapped to its cache set. An attempt to reuse a line results in a miss with a probability that depends on the cache footprint of the data accessed since the previous reference to the considered line.

Example 1. The code in Figure 2.1, which performs the addition of arrays `B` and `C` storing the result in array `A`, will be used to drive the explanation of some basic concepts of the model. The cache represented in this figure can store 16 elements with 4 sets of 1 line per set, and each line can store 4 elements. Considering this code and this cache, if the first element of array `A` is stored in the first position of a

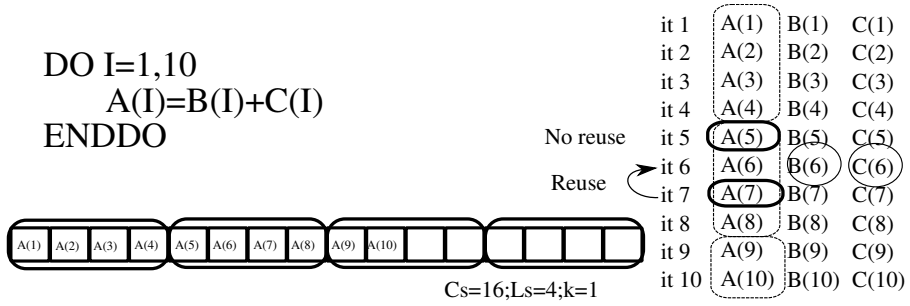


Figure 2.1: Reuse in a simple loop

memory line, then the accesses to `A(1)`, `A(5)` and `A(9)` could give place to cold misses, while the accesses to the remaining elements of the array `A` are possible interference misses ■

The PME model estimates the number of misses generated by each static reference found in a code by means of an equation, called Probabilistic Miss Equation, which includes the number of different lines it accesses (compulsory misses), the number of line reuses it generates, and the interference probability for such accesses (interference misses) during the execution of the program. Normally, each given line can be reused with different reuse distances, that is, different portions of code are executed in between different attempts to reuse the line. In the case of references found in loop nests, which is the scope of the PME model, each loop enclosing a reference gives place to a different reuse distance, which can be measured in terms of loop iterations, that (possibly) characterizes some of the reuses not captured by inner loops.

Example 2. The right side of Figure 2.1 contains the accesses to the three arrays involved in the code of the left side of the figure. The accesses belonging to each iteration are depicted in a different line. The equation that calculates the number of misses of array `A` should reflect that the accesses to `A(1)`, `A(5)` and `A(9)` are compulsory misses, while the accesses to the remaining elements of array `A` will be possible interference misses. As the access to `A` is sequential, there is a possible reuse of the line of this array accessed in the previous iteration in those accesses that are not the first ones to a cache line. The reuse distance for these possible reuses is one iteration of the loop. The data accessed since this previous access to the same line are always one element from array `B` and `C`, respectively ■

```
DO I_0 =1, N_0
  DO I_1 =1, N_1
  ...
    DO I_Z =1, N_Z
     ...
      A(f_A1(I_A1), ..., f_Aj(I_Aj)), ...)          ...
   END DO
 ...
 END DO
END DO
```

Figure 2.2: Nested loops with structures accessed using affine functions.

Our model estimates the number of misses generated by a reference by exploring the loops that enclose it from the innermost one to the outermost one. In each loop the model builds a partial PME that adds information about the reuses whose reuse distance is associated with that loop. Specifically, each partial PME estimates the number of accesses generated by the reference that cannot exploit reuse in the considered loop, the number of accesses whose reuse distance is associated with this loop, and the associated miss probability for such reuses. The PME for each loop and static reference is expressed recursively in terms of the PME for the same reference in the immediately inner loop, so that it contains all the information for the behavior of the reference within the loop. Thus, the PME associated with the outermost loop in a nest takes into account all the reuses, and its evaluation yields the number of misses generated by the reference during the execution of the loop nest.

In Section 2.2 the scope of application of the PME model is established. Section 2.3 contains a detailed description of how the miss probability of every access is calculated. This task is performed in three steps : access pattern identification, cache impact quantification and area vectors addition, which are described in Sections 2.3.1, 2.3.2 and 2.3.3 respectively. Finally, Section 2.4 describes the probabilistic miss equation that calculates the number of misses of a given reference.

C_s	Cache size
L_s	Line size
K	Associativity of the cache
D_A	# of dimensions of array `A`
$D_{\mathrm{A}j}$	size of the j-th dimension of array `A`
$d_{\mathrm{A}j}$	cumulative size of the j-th dimension of array `A`, $d_{\mathrm{A}j} = \prod_{k=1}^{j-1} D_{\mathrm{A}k}$
α_{Rj}	constant that multiplies the loop index
δ_{Rj}	constant added to a loop index
N_i	# of iterations of loop at nesting level i, whose index is $\mathtt{I}_i$
S_{Ri}	stride of reference R with respect to the loop at nesting level i, $S_{Ri} = \alpha_{Rj} \cdot d_{\mathrm{A}j}$, where j is the dimension of array `A` referenced by R indexed by $\mathtt{I}_i$
L_{Ri}	# of different sets of lines (SOLs) accessed by reference R during the execution of the loop at nesting level i

Table 2.1: Notation used in the model description

2.2. Scope of Application

Figure 2.2 depicts the scope of application of the PME model for regular access patterns. A reference to an array follows a regular access pattern when its indexes are linear functions of the loop indices, and neither indirections nor conditional statements affect the reference. The figure shows a set of normalized perfectly or non-perfectly nested loops in which the number of iterations of every loop must be the same in every execution of the loop. The reference indexes are affine functions $f_i = \alpha_i I_i + \delta_i$ of the loops control variables $\mathtt{I}_i$.

As for the hardware, our model considers set-associative caches of an arbitrary size C_s, line size L_s and associativity K with LRU replacement policy, which is the most common situation. Table 2.1 depicts these ones and other parameters we will make reference to during the explanation of our model. For simplicity, in all our terms and equations, sizes and strides are expressed in elements of the array whose access is being analyzed rather than in bytes.

2.3. Miss Probability Estimation

As explained in Section 2.1, in the study of a reference the PME model computes the number of accesses that can result in either a cold or an interference miss. The reuse distance is the distance between an access to a given line and the previous

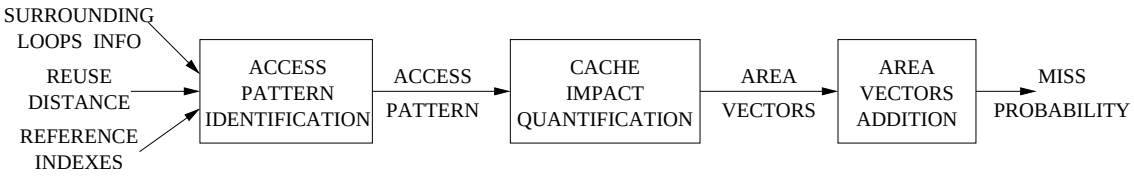


Figure 2.3: Procedure for estimating miss probabilities from the code

access to that line. It is necessary to collect the set of memory regions that has been accessed during a reuse distance. These memory regions, or, conversely, the access patterns that reference them, generate a miss probability for the attempts to reuse lines by the analyzed reference R. These probabilities are used by the PME model to estimate the number of misses of the studied reference. In a K-way set associative cache with LRU replacement policy, an attempt to reuse a line results in a miss if K or more different lines accessed since the last reference to the considered line are mapped to its cache set. As a result, the miss probability in a non-first access is equal to the probability that a cache set has received K or more lines during the reuse distance, that is, the portion of code executed since the immediately previous access to the line.

Example 3. The cache represented in Figure 2.1 is direct mapped, so it has an associativity degree $K = 1$. As a result, the miss probability in an attempt to reuse a given line is calculated as the average probability that 1 or more lines, accessed during the reuse distance, are mapped to its cache set ■

The PME model follows the three steps shown in Figure 2.3 to estimate the interference probability associated with a reuse distance:

- Access pattern identification: the access patterns followed by the references involved in the reuse distance and the parameters that characterize them are inferred from the references indexing functions and the shape of the loops that enclose them. The PME model represents each pattern as a function whose output is the footprint of the access pattern on the cache. There is one function per each typical access pattern (sequential access, access with constant stride, etc.), and its arguments provide the quantitative characterization of the access pattern.

  Example 4. In the example of Figure 2.1, in the reuse distance for the possible interference misses of array `A`, 1 iteration, there is one access to one isolated

element of arrays `B` an `C` respectively, both identified as sequential accesses to one element. ■

- Cache impact quantification: each access pattern has an associated miss probability. The lines that belong to a cache set that have received K or more lines from this pattern during the reuse distance can not be reused. So the miss probability associated to an access pattern is the ratio of sets that receive K or more lines. When the access to several arrays is considered together, it is important to keep the information about the ratio of sets that received $1 \ldots K-1$ lines, because when the effects of these lines from different arrays are considered together they can contribute to increase the miss probability. So, a vector of probabilities, called *Area Vector*, is associated to each access pattern.

  Example 5. In the example of Figure 2.1, the cache can store $C_s = 16$ elements distributed in 4 sets where every set stores $K = 1$ cache line of $L_s = 4$ elements. In the reuse distance one element of arrays `B` and `C` is accessed, each one of these element will go to 1 of the 4 cache sets. The destination set will be determined by the base position of the arrays. It is known that for each array, `B` and `C`, 1 of the 4 cache sets will receive $K = 1$ lines while the remaining 3 cache sets will not receives any element from the array. So, a cache set receive 1 line with a probability $1/4 = 0.25$, and that is the miss probability associated to that reuse distance for the access to that array ■

- Area vectors addition: once the area vectors for the different access patterns have been estimated, they must be added in order to calculate a global area vector that represents their summarized impact on the cache.

Once these three steps are completed, the final interference probability is estimated as the ratio of sets that received K or more lines during the reuse distance, which is conversely the probability a given set has received K or more lines. This value can be extracted from the global area vector associated with the analyzed reuse distance. We will now describe in more detail the three steps of the miss probability estimation process.

2.3.1. Access Pattern Identification

In Section 2.1 we saw that reuse distances are measured in terms of the number of iterations of a loop. In order to identify the access pattern that a given reference R follows during a reuse distance consisting of n iterations of the loop at nesting level h, the indexes of each dimension and the number of iterations of each loop during this reuse distance are examined. The output of this analysis is a D_{A}-tuple $\mathcal{R}_R(h,n)$, where D_{A} is the number of dimensions of the array `A` referenced by R. Each element of this tuple consists in its turn of a 2-tuple $R_{Rj} = (M_j, S_j)$, where the M_j is the number of different points accessed along dimension j and S_j the constant stride between two consecutive points.

The algorithm followed to calculate the 2-tuple associated to dimension j of reference R during n iterations of the loop at nesting level h is described now. When the index of the reference is an affine function $\alpha_{Rj} \cdot \mathtt{I}_i + \delta_{Rj}$ of some loop index $\mathtt{I}_i$, the set of points accessed in this dimension by R can be represented as the tuple $(\mathrm{Iters}_i(h,n), S_{Ri})$, where $\mathrm{Iters}_i(h,n)$ is the number of different values that $\mathtt{I}_i$ takes during n iterations of the loop in nesting level h. This value is calculated as

$$\mathrm{Iters}_i(h,n) = \begin{cases} 1 & \text{if } i < h \\ n & \text{if } i = h \\ N_i & \text{if } i > h \end{cases}$$

Let us remember that the loops are labeled from the outermost one, at nesting level 0, to the innermost one using increasing integer values. The value S_{Ri} is the stride that reference R has with respect to loop i. This stride is a constant, since the index we are considering is an affine function of $\mathtt{I}_i$. S_{Ri} is calculated as $\alpha_{Rj} \cdot d_{\mathrm{A}j}$, where j is the dimension whose index depends on $\mathtt{I}_i$; α_{Rj} is the scalar that multiplies the loop variable in the affine function, and $d_{\mathrm{A}j}$ is the cumulative size[1] of the j-th dimension of the array `A` referenced by R.

Once the D_{A}-tuple $\mathcal{R}_R(h,n)$ that represents the region of array `A` accessed by R during n iterations of the loop at nesting level h has been calculated, some simplifications may be applied between pairs of 2-tuple R_{Rj} that describe the access

[1] Let `A` be an N-dimensional array of size $D_{\mathrm{A}1} \times D_{\mathrm{A}2} \times \ldots D_{\mathrm{A}N}$, we define the cumulative size for its j-th dimension as $d_{\mathrm{A}j} = \prod_{i=1}^{j-1} D_{\mathrm{A}i}$

```
DO I = 1, M
  DO K = 1, N
    DO J = 1, H
     C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

Figure 2.4: Matrix Product

pattern in different dimensions of the array:

$$\begin{aligned} ((1, S_j), (M_k, S_k)) &= (M_k, S_k) \\ ((M_j, S_j), (M_k, M_j \cdot S_j)) &= (M_j \cdot M_k, S_j) \end{aligned}$$

After these simplifications a single 2-tuple (M_s, S_s) that describes the region accessed by the reference is typically obtained.

Rather than this description of the memory region accessed, the output of the access pattern identification step is a function that characterizes the access pattern whose output is the area vector associated to it. Depending on the values of S_s in a tuple R_{Rj}, two kinds of access pattern functions can be identified:

1. If $S_s = 1$, it is an access to M_s consecutive elements. We denote the function that calculates the area vector associated to a region of M_s consecutive elements as $\mathrm{Reg_s}(M_s)$.

2. Otherwise it is an access to a set of M_s regions of one element separated by a constant stride S_s. Such access pattern is represented by the function $\mathrm{Reg_r}(M_s, 1, S_s)$.

Finally, although the access pattern functions have been presented based on the values of a single tuple R_{Rj}, it is not always possible to reduce $\mathcal{R}_R(h, n)$ to a single tuple. All the cases of this kind we have found in the codes we have analyzed had the form $\mathcal{R}_R(h, n) = ((M_1, 1), (M_2, S_2))$, which can be represented by function $\mathrm{Reg_r}(M_2, M_1, S_2)$. It represents an access to M_2 separate groups of M_1 consecutive elements separated by a constant stride S_2.

Example 6. We will use the code in Figure 2.4 as a driving example to illustrate the different steps of the PME model. This code performs the product between two matrices `A` and `B` and stores the result in matrix `C`. The calculation of the miss probability associated to an access whose reuse distance is one iteration of loop `K` must consider the effects of all the accesses that take place during that reuse distance. In one iteration of this loop there are accesses to arrays `A`, `B` and `C`:

- Reference `C(I, J)`: The first dimension of reference `C(I, J)` is indexed by the index of the outermost loop `I` at nesting level 0, so the tuple R_{R1} that describes its access is $(\text{Iters}_0(1,1), S_{R1})$, being $\text{Iters}_0(1,1) = 1$ and $S_{R1} = 1$. The second dimension is indexed by the index of the innermost loop `J` at nesting level 2 so $R_{R2} = (\text{Iters}_2(1,1), S_{R2})$ being $\text{Iters}_2(1,1) = N_2 = H$ where N_2 is the number of iterations of the loop at nesting level 2, and $S_{R2} = M$. So, $\mathcal{R}_R(1,1)$, the pair of tuples that define the access to each dimension of the array `C` in 1 iteration of nesting level 1, is $((1,0),(H,M))$ which can be simplified to the tuple (H,M). This tuple will be identified as a region $\text{Reg}_\text{r}(H,1,M)$, that is, H groups of 1 element separated by a distance M.
- Reference `A(I,K)`: The first dimension in reference `A(I,K)` is indexed also by the index of the outermost loop `I`, so the tuple that characterizes the access in this dimension is also $(1,0)$. The second dimension is indexed by the index of the current loop `K`, so $\text{Iters}_1(1,1) = n = 1$ and $S_{R1} = H$, resulting in the tuple $(1,H)$. These two tuples can also be simplified to the tuple $(1,H)$, which can be identified as a region $\text{Reg}_\text{s}(1)$ the access to one element of this data structure.
- Reference `B(K, J)`: This reference is indexed by the index of loop `K` in its first dimension, so the associated tuple is $(1,1)$, while the second dimension is indexed by the index of the inner loop `J`, so $Iters_2(1,1) = H$ and $S_{R2} = N$, resulting in the tuple (H,N). Both tuples can be merged and the resulting tuple is (H,N), which can be identified as a region $\text{Reg}_\text{r}(H,1,N)$, the access to H groups of 1 element separated by a distance N ■

2.3.2. Cache Impact Quantification

The functions identified in the previous step are evaluated in order to yield vectors of probabilities called area vectors that represent the impact on the cache of the access they represent. The area vector V associated with a given set of accesses

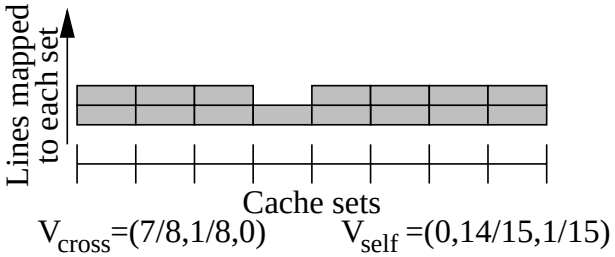


Figure 2.5: Cross and sef interference area vectors for a footprint on a 2-way associative cache with 8 sets

on a cache with associativity K consists of $K+1$ probabilities $V_0, V_1, \ldots, V_K$. The PME model considers two kinds of area vectors:

- **Cross interference area vectors** represent the impact on the cache of the considered access pattern as viewed by lines not involved in the access. In these vectors, $V_i, K \geq i > 0$ is the ratio of sets that hold $K - i$ lines of the accessed region, and V_0 is the ratio of sets that hold K or more lines. These ratios are also conversely the probabilities. For example V_0, is the probability that a set in the cache has received K or more lines accessed by the pattern, V_1 is the probability a cache set has received $K - 1$ lines, and so on.
- **Self interference area vectors** represent the impact of the footprint on the probability of reuse for the lines it involves. In these vectors, V_0 is the probability that a line of the footprint is competing in its cache set with other K or more lines of the footprint. For $K \geq i > 0$, V_i is the probability a line of the footprint shares its cache set with other $K - i$ lines of the access.

Example 7. As an example let us consider a 2-way associative cache with eight sets and a reference that has just accessed 15 lines sequentially. As a result, seven of the eight sets contain two of the lines referenced, while the other set contains just one line, as it is illustrated in Figure 2.5. The cross interference area vector generated by this access is $(7/8, 1/8, 0)$, as 7 out of the 8 sets have received two or more lines from the access; only one set received a single line, and no sets received zero lines. These ratios are conversely the probabilities a randomly chosen set has two or more, one, or zero lines in it, respectively.

The self interference area vector for this access is $(0, 14/15, 1/15)$. The first component is zero, as none of the lines involved in the access has to compete for

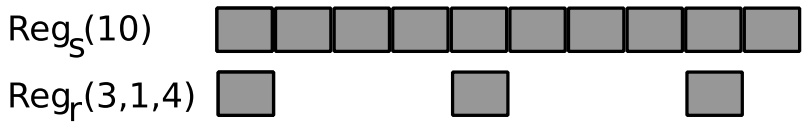


Figure 2.6: Footprints of the most common regular access patterns

its cache set with other two or more other lines from the footprint. The second component is the ratio of lines of the footprint that share their cache set with exactly one line (14 out of 15). Finally, according to the third component, only one of the 15 lines of the footprint does not share its set with any other line of the footprint. These ratios are conversely the probabilities a randomly chosen line of the footprint has to compete in its set with two or more, one, or no lines, respectively ■

The equations and algorithms developed to estimate the cross and the self interference area vectors associated to the sequential access and the access with a constant stride access patterns are presented now.

Sequential access to n consecutive words $\mathbf{Reg_s}(n)$

The sequential access to n consecutive words $\mathrm{Reg_s}(n)$ (see Figure 2.6) generates a cross interference area vector AV_s:

$$\begin{array}{l} AV_{\mathrm{s}_{(K-\lfloor l \rfloor)}}(n) = 1 - (l - \lfloor l \rfloor) \\ AV_{\mathrm{s}_{(K-\lfloor l \rfloor - 1)}}(n) = l - \lfloor l \rfloor \\ AV_{\mathrm{s}_i}(n) = 0 \quad 0 \leq i < K - \lfloor l \rfloor - 1, K - \lfloor l \rfloor < i \leq K \end{array} \tag{2.1}$$

where $l = \max\{K, (n + L_\mathrm{s} - 1)/(L_\mathrm{s} N_K)\}$ is the maximum of K and the average number of lines placed in each set. In this expression, L_s stands for the line size and N_K for the number of cache sets. The number of cache sets N_K can be calculated as $C_\mathrm{s}/L_\mathrm{s}K$. The term $L_\mathrm{s} - 1$ added to n stands for the average extra words brought to the cache in the first and last accessed lines.

Expression $AV_s(C(n)C_{sk})$ calculates the autointerference area vector of this access, being $C_{sk} = C_s/K$. The autointerference that affects to each line is equal to the cross interference of an array of $C(n)(C_{sk})$ elements, where $C(n)$ is the number of lines of the array competing by the same cache set. This array would add $C(n)$ lines to each set. The method to calculate $C(n)$ is detailed in the next section.

Number of lines of a vector competing for the same cache set

We need a function to compute the average number of different lines that compete with a given line for the same cache set. This value will be used in the calculation of the self-interference probability. For a data structure of n consecutive words, this function is defined as:

$$\begin{aligned} C(n) &= \lfloor v \rfloor \frac{(v - \lfloor v \rfloor)(\lfloor v \rfloor + 1)}{v} + (\lfloor v \rfloor - 1)\left(1 - \frac{(v - \lfloor v \rfloor)(\lfloor v \rfloor + 1)}{v}\right) \\ &= \frac{\lfloor v \rfloor}{v}(2v - \lfloor v \rfloor - 1) \end{aligned} \tag{2.2}$$

where $v = n/C_{\mathrm{sk}}$ is the average number of the data structure associated to a given cache set. If $v > 1$, in the $(v - \lfloor v \rfloor) \times 100$ % of the cache sets, the number of lines of the data structure competing with another one is $\lfloor v \rfloor$. In this area the $\frac{(v-\lfloor v \rfloor)(\lfloor v \rfloor+1)}{v} \times 100$ % lines of the data structure are located. In the remaining cache, the number of lines that competes is $\lfloor v \rfloor - 1$. Using this average value, the equation (2.2) is obtained. Besides it can be checked that if $v \leq 1$, that is, when the data structure covers a number of lines minor or equal than the number of cache sets, this expression takes the value 0, because there is no self-interference.

Access with a constant stride $\mathbf{Reg_r}(N_\mathbf{r}, T_\mathbf{r} S_\mathbf{r})$

The estimation of the area vector associated to an access to several elements of the data structure with a constant stride is performed through a mixed method that involves the calculation of the starting and ending points of each region on the cache, from which we obtain the arithmetic mean of the number of lines mapped to each cache set. The corresponding area vector is obtained from these values.

Let us consider the access to N_r regions of size T_r with a constant stride S_r between two consecutive regions $\mathrm{Reg_r}(N_\mathrm{r}, T_\mathrm{r}, S_\mathrm{r})$(see Figure 2.6). In the first step the positions C_i and F_i corresponding to the start and the end of each region in the cache are calculated, considering that:

$$\begin{aligned} &C_0 = 0 \\ &C_i = (C_{i-1} + S_\mathrm{R}) \bmod (C_\mathrm{s}/K), 0 < i \leq N_\mathrm{R} \\ &F_i = (C_i + T_\mathrm{R} - 1) \bmod (C_\mathrm{s}/K), 0 \leq i \leq N_\mathrm{R} \end{aligned} \tag{2.3}$$

where C_s is the cache size and K is the degree of associativity. From now on

$C_{\mathrm{sk}} = C_{\mathrm{s}}/K$. In two vectors CV and FV of size C_{sk}, initialized to zero, we add one unit for each position associated with a C_i or an F_i, respectively. They are then analyzed calculating the average number of lines of the access corresponding to each set of L_{s} positions in these vectors, that is to say, to a line of a cache set. Three values are used to do this. The first one is given by:

$$L_{\mathrm{G}}(0) = \lfloor (T_{\mathrm{r}} - 1)/C_{\mathrm{sk}} \rfloor N_{\mathrm{r}} + \sum_{i=C_{\mathrm{sk}}-(T_{\mathrm{r}}-1) \bmod C_{\mathrm{sk}}}^{C_{\mathrm{sk}}-1} \mathrm{CV}(i) \tag{2.4}$$

that stands for the number of lines corresponding to different regions that are guaranteed to be associated to the first set in the considered cache. These lines come from all of the regions if $T_{\mathrm{r}} \geq C_{\mathrm{sk}}$, which is the first term in the addition; and/or from those regions that start in preceding sets and whose end has not been reached. For a set starting in position j this value is recalculated as:

$$L_{\mathrm{G}}(j) = L_{\mathrm{G}}(j-1) + \mathrm{CV}(j-1) - \mathrm{FV}(j-1) \tag{2.5}$$

On the other hand, we have $L_{\mathrm{F}}(j)$, the average number of lines associated to regions that end in the set starting in position j of the cache having chosen $C_0 = 0$, but that with shifts $C_0 = 1, \ldots L_{\mathrm{s}} - 1$ might finish in the next set. It is calculated as:

$$L_{\mathrm{F}}(j) = \sum_{i=j}^{j+L_{\mathrm{s}}-1} \mathrm{FV}(i)(i-j)/L_{\mathrm{s}} \tag{2.6}$$

The number of regions that start in a given cache set must be taken into account to calculate the average number of lines associated to it. This requires using a weight similar to the one used in (2.6) to take into account the possibility that with different starting positions for C_0 the regions start in the next set. This value L_{C} would be calculated for each set starting in position j as:

$$L_{\mathrm{C}}(j) = \sum_{i=j}^{j+L_{\mathrm{s}}-1} \mathrm{CV}(i)(L_{\mathrm{s}} - (i-j))/L_{\mathrm{s}} \tag{2.7}$$

We are now in a position to calculate the average number of lines associated to the cache sets starting in positions $j = 0, L_{\mathrm{s}}, \ldots, C_{\mathrm{sk}} - L_{\mathrm{s}}$ as:

$$L(j) = L_{\mathrm{G}}(j) + L_{\mathrm{F}}((j + C_{\mathrm{sk}} - L_{\mathrm{s}}) \bmod C_{\mathrm{sk}}) + L_{\mathrm{C}}(j) \tag{2.8}$$

Finally, the cross interference area vector associated to this access would be calculated from these values as:

$$AV_{\mathrm{r}}(N_{\mathrm{r}}, T_{\mathrm{r}}, S_{\mathrm{r}}) = \frac{1}{N_K} \sum_{i=0}^{N_K-1} AV_s(L(iL_{\mathrm{s}})C_{\mathrm{sk}}) \tag{2.9}$$

because an average of $L(iL_{\mathrm{s}})$ lines will be mapped to the i-th cache set, and the cross interference area vector associated to an interference with n different lines is $AV_s(nC_{\mathrm{sk}})$.

The calculation of the self-interference area vector is performed as follows:

$$AV_{\mathrm{ra}}(N_{\mathrm{r}}, T_{\mathrm{r}}, S_{\mathrm{r}}) = \frac{\sum_{i=0}^{N_K-1} AV_s(\max\{0, L(iL_{\mathrm{s}}) - 1\}C_{\mathrm{sk}})L(iL_{\mathrm{s}})}{\sum_{i=0}^{N_K-1} L(iL_{\mathrm{s}})} \tag{2.10}$$

The same idea is applied but considering that each line in the i-th set competes with other $L(iL_{\mathrm{s}})-1$ lines. The self-interference area vector for each set is multiplied by the number of lines that go to that set, obtaining the final vector averaged by line.

2.3.3. Area Vectors Addition

The preceding step generates an area vector per data structure accessed during a reuse distance. Each component of one of these area vectors V yields the probability a given cache set will hold K or more (V_0), or $K-1$ (V_1), etc. lines because of the accesses to the corresponding data structure that can interfere with the reuses of the reference whose behavior is being analyzed. In this final step of the process these area vectors are added in order to get a global interference area vector that represents the total impact on the cache of all the accesses that take place during the considered reuse distance. The component 0 of this global area vector is the miss probability we are trying to estimate. Given two area vectors V_{A} and V_{B}, their addition, represented by the operator $\cup$, is calculated as

$$\begin{aligned} (V_{\mathrm{A}} \cup V_{\mathrm{B}})_0 &= \textstyle\sum_{j=0}^{K} \left(V_{\mathrm{A}_j} \sum_{i=0}^{K-j} V_{\mathrm{B}_i} \right) \\ (V_{\mathrm{A}} \cup V_{\mathrm{B}})_i &= \textstyle\sum_{j=i}^{K} V_{\mathrm{A}_j} V_{\mathrm{B}_{(K+i-j)}} \qquad 0 < i \leq K \end{aligned} \tag{2.11}$$

This method is based on the addition as independent probabilities of the area ratios, which means that it does not take into account the relative positions of the program data structures in memory. This approach allows the PME model to provide reasonable estimations in many situations in which the base addresses of the data structures are not known at compile time (e.g. physically-addressed caches, dynamically allocated data structures, ...), something that, as far as we know, no other model supports. When those base addresses are known at compile time, each area vector is scaled before its addition by means of a coefficient that represents the amount of overlapping between the region it represents and the data structure associated to the reference whose PME is being calculated in the cache. See [31] for more details.

2.4. Building Probabilistic Miss Equations

A partial PME F_{Ri} is built for each static reference R in the code and loop at nesting level i that encloses such reference. This PME estimates the number of misses that R generates during a complete execution of this loop as a summatory of the number of accesses that enjoy each possible reuse distance associated with this loop multiplied by the miss probability that the memory regions accessed during that reuse distance generate. Of course every access that is the first one to a line in this loop, cannot result in reuses of lines already accessed in the current execution of the loop, thus their miss probability cannot be associated to reuse distances within the loop. The miss probabilities for those accesses correspond either to (a) reuse distances that are associated with outer loops; or (b) reuse distances with respect to accesses to the same data in previous loops in the same nesting level, when we consider non-perfectly nested loops; or (c) when the loop is the outermost one ($i = 0$) and there are no preceding loops that could give place to reuses, the miss probability is simply one, since every first access to a line in this loop is indeed a first access to the line, unable to exploit any reuse, which results in a compulsory miss. Since PMEs are built beginning in the innermost loop and proceeding outwards, and their evaluation depends on memory regions associated with reuses that are calculated in outer or previous loops, the general expression of a PME is F_{Ri}(RegIn), where RegIn stands for the memory regions accessed during the reuse distance for what in this level of the nest happen to be first accesses. The exception are the PMEs for outermost loops F_{R0}, in which no reuse from previous accesses is possible. For their evaluation we use as RegIn a memory region whose associated miss probability

is one, so that the first-time accesses to a line in the nest are predicted as misses. In general we can define the input parameter RegIn of a PME F_{Ri} as the memory region accessed since the immediately previous access to any of the lines that R references in loop i in the moment the execution of the loop begins.

If the variable $\mathtt{I}_i$ associated with loop i does not index the array `A` or it indexes it directly across an affine function, the access pattern of R is regular with respect to loop i. Thus, the behavior of R in this nesting level is modeled by the regular access PME explained in [31]:

$$F_{Ri}(\text{RegIn}) = L_{Ri} \cdot F_{R(i+1)}(\text{RegIn}) + (N_i - L_{Ri}) \cdot F_{R(i+1)}(\text{Reg}_{Ri}(1)) \ , \qquad (2.12)$$

where N_i is the number of iterations of the loop at the nesting level i, and L_{Ri} is the number of iterations in which there is no possible reuse for the lines referenced by R from the point of view of this loop. $\text{Reg}_{Ri}(j)$ stands for the memory region accessed during j iterations of the loop in the nesting level i that can interfere with the accesses of R in the cache.

The equation calculates the total number of misses for reference R in nesting level i as the sum of two values. The first one is the number of misses produced by the L_{Ri} iterations in which the accesses of R cannot exploit reuse in this loop. The miss probability for these iterations depends on reuse distances generated in outer or preceding loops, thus the number of misses generated in these iterations is obtained evaluating $F_{R(i+1)}$, the PME for the immediately inner loop, passing as parameter for the calculation of the miss probability of its first accesses the value RegIn provided by those external loops. The second value corresponds to the iterations in which there can be reuse with respect to the accesses in the previous iteration in this loop. The miss probability for the first accesses in the evaluation of the PME for the immediately inner level depends in this case on the memory regions accessed during one iteration of loop i.

When this equation is applied to the innermost loop containing reference R the end of the recursivity is achieved substituting $F_{R(i+1)}(Reg)$ by $AV_0(Reg)$, that is, the miss probability associated with region Reg. In the innermost loop these L_{Ri} iterations correspond to lines: they mean that during one complete execution of the N_i iterations of the innermost loop, R really accesses L_{Ri} different lines, the other accesses being thus reuses. When the loop analyzed is not the innermost one, the iterations of the loop define sets of lines (SOLs) accessed by R in the inner loops. For example, if a bidimensional $M \times N$ FORTRAN array is accessed row by row

(that is, the innermost loop of the access sweeps through the N columns of a given row), in the analysis of the outer loop that controls the row index of the reference, each iteration of this loop is associated to the access to the set of lines that hold the elements of a row of the matrix. As FORTRAN stores the arrays by columns, if $M \geq L_s$, where L_s is the cache line size measured in elements, which is the most usual situation, each set of lines will be made up of N different lines. In this case, L_{Ri} iterations of this outer loop give place to accesses to new sets of lines (SOLs); while the other $N_i - L_{Ri}$ iterations generate reuses of the SOLs accessed in the previous iteration. In what follows we will talk in general about sets of lines (SOLs), in the understanding that in the innermost loop each one of these sets consists of a single line.

The number of iterations of loop i that cannot exploit either spatial or temporal locality is given by

$$L_{Ri} = 1 + \left\lfloor \frac{N_i - 1}{\max\{L_s/S_{Ri}, 1\}} \right\rfloor \ , \tag{2.13}$$

where L_s is the line size measured in elements of the array referenced by R and S_{Ri} is stride that reference R has with respect to loop i.

Example 8. In the driving example of Figure 2.4 the reference `B(K, J)` is contained in the innermost loop `J` at nesting level 2. In this nesting level the loop index `J` indexes the second dimension using the affine function 0+`J`. The number of different lines of `B` accessed, L_{R2}, is calculated using the Equation 2.13, being $N_2 = H$ and $S_{R2} = d_{A2} = N$, so $L_{R2} = H$ assuming that $L_s \leq N$. The resulting equation for this nesting level is $F_{R2}(\text{RegIn}) = H \cdot F_{R3}(\text{RegIn})$. As it is the innermost level $F_{R3}(\text{RegIn}) = AV_0(\text{RegIn})$.

In the nesting level 1, the loop index `K` indexes the first dimension of `B` using the affine function 0+`K`. $S_{R1} = d_{A1} = 1$ and $N_1 = N$, so the number of different SOLs accessed is $L_{R1} = 1 + \lfloor (N-1)/L_s \rfloor$. The final equation for this nesting level is

$$F_{R1}(\text{RegIn}) = (1+\lfloor (N-1)/L_s \rfloor) \cdot F_{R2}(\text{RegIn}) + (N - (1+\lfloor (N-1)/L_s \rfloor)) \cdot F_{R2}(\text{Reg}_{R2}(1))$$

In the outermost level the loop index does not index any dimension of the array `B`. $S_{R0} = 0$ and $N_0 = M$, so $L_{R0} = 1$ and the final equation that characterizes the access in this nesting level is $F_{R0}(\text{RegIn}) = F_{R1}(\text{RegIn}) + (M-1) \cdot F_{R1}(\text{Reg}_{R0}(1))$ ■

Chapter 3

Model Extension to Handle Codes with Conditional Statements

The original PME model described in the previous chapter can only analyze codes with regular access patterns. This thesis covers the extension of the PME model to model irregular access patterns. Different extensions are proposed depending on whether the irregularity is due to the presence of data-dependent conditional statements or indirections. This chapter contains a description of the PME model extension for codes with irregular access pattern due to data-dependent conditional statements.

Data-dependent conditional statements are a significant subset of the conditional structures whose outcome depends on computations made at run-time, and where the pattern of the condition is highly irregular. These statements are not statically analyzable and their truth values can not be determined at compile time, that is, it can not determined if the conditional statement will be true or false in each one of their evaluations. Furthermore, their truth values can change between different executions of the program if the input data vary. In this PME model extension we will consider codes with any kind and number of conditional sentences, even with references and whole loop nests controlled by several nested conditionals, and nested in any arbitrary way. Only two restrictions are set on the conditions. The first one is that their verification must follow an uniform distribution. The second one is that the conditions must be independent, that is, the probability that a given condition is fulfilled is not influenced by the fact that any other condition(s) are fulfilled or not.

In regular codes there is a statically determinable sequence of accesses associated to a reference in a nesting level. These accesses generate a series of possible reuses of lines accessed in previous iterations of the loop or previous loops. The miss probability measures the probability that each reuse attempt results in a miss using the probabilistic nature of the PME model. When one or more conditional statements guard a reference in the code, there is a sequence of potential accesses associated to this reference. In this case, each access takes place only when the conditions of the conditional statements that guard the reference are fulfilled. So, there are different possible reuse distances and each one of them has its associated miss probability. The probabilistic capabilities of the PME model are used for determining the probability of each reuse distance using the probability that each potential access generated by the reference actually takes place.

Some extensions are required to consider irregular accesses due to conditionals. One is the identification of new access patterns that give place to footprints not considered by the original PME model, and for which new methods must be developed in order to estimate their corresponding area vectors. Some steps of the miss probability estimation process need also some adaptations to cover these new situations. A new kind of PMEs is also needed. In these PMEs reuses take place only with a given probability, and their reuse distance varies depending on the behavior of the conditional sentences found in the nest.

Section 3.1 describes the extended scope of application of the PME model. In Section 3.2, the miss probability estimation process is adapted to cover also references guarded by conditional statements. Section 3.3 describes a new type of PME equation to characterize the cache behavior of codes guarded by conditionals. Finally, Section 3.4 contains the results of the validation of this model extension.

3.1. Scope of Application

The scope of application of the extended model is shown in Fig. 3.1. We now consider any number of arbitrarily nested conditional statements, with an arbitrary number of atomic conditions that involve any number of data elements. The figure only shows one data element per condition for simplicity. The `IF` statements condition the execution of isolated references or complete loops or nests. The restrictions in the PME model of constant number of loop iterations and affine indexing continue to hold. Also, our current systematic strategy to model irregular access patterns

```
DO I_0=1, N_0, L_0
  DO I_1=1, N_1, L_1
  ...
  IF cond(D(f_D1(I_D1), ..., f_DdD(I_DdD)))
  ...
    DO I_Z=1, N_Z, L_Z
      A(f_A1(I_A1), ..., f_AdA(I_AdA))
      ...
      IF cond(B(f_B1(I_B1), ..., f_BdB(I_BdB)))
        C(f_C1(I_C1), ..., f_CdC(I_CdC))
      ...
    END DO
  ...
  END DO
END DO
```

Figure 3.1: Loop nest with data-dependent conditional statements.

requires the conditions in the code to follow an uniform distribution and to be independent. This latter restriction means that the probability that a given condition is fulfilled does not depend on the verification of other conditions in the code. The different conditions may be fulfilled with different probabilities each.

3.2. Miss Probability Estimation in Irregular Codes

In Section 2.3, three different steps were described to estimate the miss probability associated to a given reuse distance: access pattern identification, cache impact quantification and area vectors addition. Some changes must be done in the access pattern identification and cache impact quantification steps to cover the existence of conditional statements in the code. The area vectors addition step does not need any adaptation.

3.2.1. Access Pattern Identification

In Section 2.3.1 we established that in order to identify the access pattern that a given reference R follows during a reuse distance consisting of n iterations of the loop at nesting level h, the indexes of each dimension and the number of iterations of

each loop during this reuse distance are examined. The output of this analysis is a D_{A}-tuple $\mathcal{R}_R(h,n)$, where D_{A} is the number of dimensions of the array `A` referenced by R. Each element of this tuple consisted in its turn of a 2-tuple $R_{Rj} = (M_j, S_j)$, where the M_j is the number of different points accessed along dimension j and S_j the constant stride between two consecutive points. This method allowed to describe any regular access pattern on an array indexed using affine functions of the loop indices. If the array is guarded by a conditional statement, we need an additional output to this analysis that is the probability $P_R(h,n)$ that each element of the array is accessed during n iterations of the loop of nesting level h.

This probability $P_R(h,n)$ depends not only on the access pattern of the reference in this nesting level, but also in the inner ones. As a result, its calculation takes into account all the loops from the h-th down to the one containing the reference. If fact, this probability is calculated recursively in the following way:

- If h is the innermost loop containing R, then $P_R(h,n) = p_h$ being p_h the product of all the probabilities associated to the conditional sentences controlling R in nesting level h.
- If h is not the innermost loop containing R and the loop index is not used in the references found in conditions that control R or does not index any dimension of the array accessed in R, then $P_R(h,n) = p_h P_R(h+1, N_{h+1})$, being N_{h+1} the number of iterations of the loop $h+1$.
- Otherwise, $P_R(h,n) = \overline{\overline{p_h P_R(h+1, N_{h+1})}}^{n}$.

The same rules used in 2.3.1 can be used to reduce the D_{A}-tuple $\mathcal{R}_R(h,n)$ to an unique tuple (M_s, S_s) that describes the whole access. Using this formal description of the memory region accessed, a function must be obtained that characterizes the access pattern and whose output is the area vector associated to it. This process must be also extended to cover the irregular access patterns produced by the presence of conditional statements. In the case of references with regular access patterns, $P_R(h,n) = 1$, the translation remains the same as the one explained in Section 2.3.1. But when the probability $P_R(h,n) < 1$, as each point involved in the pattern has only a certain probability of being actually accessed, the following rules are applied.

1. If $S_s = 1$, it is an access to M consecutive elements in which each element is accessed with a probability $P_R(h,n)$. The function that calculates the area vector for this access is $\mathrm{Reg}_{\mathrm{sp}}(M_s, P_R(h,n))$.

```
DO I = 1, M
  DO K = 1, N
    IF (A(I,K) .NEQ. 0)
      DO J = 1, H
        IF (B(K,J) .NEQ. 0) THEN
          C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ENDIF
      ENDDO
    ENDIF
  ENDDO
ENDDO
```

Figure 3.2: Optimized product of matrices

2. Otherwise the access affects M_s different points separated by a constant stride S_s, which each element is accessed with a probability $P_R(h,n)$. The area vector associated to this access pattern is estimated by function $\mathrm{Reg}_{\mathrm{rp}}(M_s, 1, S_s, P_R(h,n))$.

As we see, the existence of conditional accesses define probabilistic counterparts for $\mathrm{Reg}_{\mathrm{s}}$ and $\mathrm{Reg}_{\mathrm{r}}$, that characterize those access patterns in which each element is accessed with a certain probability. The most general function is $\mathrm{Reg}_{\mathrm{rp}}$, all the other ones being specializations of this one. Similarly, $\mathrm{Reg}_{\mathrm{s}}$ functions are specializations for $S = 1$ of their $\mathrm{Reg}_{\mathrm{r}}$ counterparts, and the area vector functions that depend on a probability of access P yield the same output as their regular counterparts for $P_R(h,n) = 1$. Still, we find this distinction useful because regular access patterns enable simpler and faster algorithms for the calculation of their associated area vector than irregular access patterns, and the same happens with the $\mathrm{Reg}_{\mathrm{s}}$ functions with respect to their $\mathrm{Reg}_{\mathrm{r}}$ counterparts with input stride one.

Sometimes $\mathcal{R}_R(h,n)$ can not be reduced to a single tuple. All the cases of this kind we have found in the codes we have analyzed had the form $\mathcal{R}_R(h,n) = ((M_1,1),(M_2,S_2))$, which can be represented by function $\mathrm{Reg}_{\mathrm{rp}}(M_2, M_1, S_2, P_R(h,n))$, as they are an access to M_2 separate groups of M_1 consecutive elements each which are separated by a constant stride S_2, in which each individual element of the region has a probability $P_R(h,n)$

Example 9. The code in Figure 3.2 implements the product of two matrices, `A` and `B`, which may have many zero entries. As an optimization, when the element of `A` to be used in the current product is 0, then all its products with the corresponding

elements of B are not performed. Also, if the element of B to be used in the current product is 0 then that operation is not performed either. This avoids two floating point operations and the load and storage of C(I,J). The innermost conditional statement has an uniform probability p_2 of being fulfilled while the outermost one has a probability p_1.

Just as in the modeling of the code of Figure 3.2, without loss of generality, we assume a compiler that maps scalar variables to registers and which tries to reuse the memory values recently read in processor registers. Under these conditions, the code in Figure 3.2 contains three reference to memory: C(I, J),A(I, K) and B(K, J). The first dimension of array C is indexed by the index of the outermost loop 0 using the affine function 0+I, so $\text{Iters}_0(0,1) = 1$, $S_{R1} = d_{\text{A}1} = 1$ resulting in the tuple $(1,1)$. The second dimension is indexed by the index of the innermost loop using the affine function 0+J, $\text{Iters}_0(0,1) = N_2 = H$, $S_{R2} = d_{\text{A}2} = M$ resulting in the tuple (H, M). These two tuples will be simplified to the tuple (H, M). About the calculation of the probability $P_R(0,1)$ of accessing each element of array C in one iteration of level 0, in that level, the reference is affected by the loop index I so $P_R(0,1) = 1 - (1 - p_1 P_R(1,1))^N$ because $N_1 = N$. In the inner level 1, the loop index K does not affect to the reference C(I, J), so $P_R(1,1) = P_R(2,1)$ as $p_1 = 1$. Level 2 is the innermost level containing that reference and $P_R(2,1) = p_2$. So, $P_R(0,1)$ can be calculated as $1 - (1 - p_1 p_2)^N$ This will be mapped as an access $\text{Reg}_{\text{rp}}(M, 1, H, P_R(0,1))$ to M regions of 1 element separated by a distance H where each element has a probability $1 - (1 - p_1 p_2)^N$ of being accessed.

The first reference to array A, A(I, K), is located inside loop K. There is a second reference in the innermost loop that will not produce a new memory access because it is considered to be satisfied from the processor registers. The first dimension of this reference is indexed by the loop index of the outermost loop, so the tuple that describes the access is $(1,1)$. The second dimension is indexed by the loop index of the inner loop K so the tuple for this dimension is (N, M). These two tuples can be merged in the tuple (N, M) that is mapped to an access $\text{Reg}_{\text{r}}(N, 1, M)$, N groups of 1 element separated by a distance M.

The reference B(K, J) is contained in the innermost loop. The tuples for the first and second dimension are $(N, 1)$ and (H, N) respectively. The probability $P_R(0,1)$ that each element this reference could access is actually accessed is p_1. They can not be simplified to an unique tuple but it can be identified as an access $\text{Reg}_{\text{rp}}(H, N, N, p_1)$, access to H groups of N elements separated by a distance N. This region can be identified as the special case of $\text{Reg}_{\text{sp}}(HN, p_1)$ the access to HN

consecutive elements with a given probability ■

3.2.2. Cache Impact Quantification in Irregular Codes

The two access patterns usually found in codes with regular access that were described in Section 2.3.2 are the sequential access and the access to groups of consecutive elements of the same size that are separated by a constant stride. Their irregular counterparts, when uniform probabilities of access are considered, are described in a similar way, with the important difference that now each one of the elements involved in the pattern is accessed with a given probability p that is the same one for each element. The modeling of these new access patterns, which we detail below, depends on the cache parameters. Let us remember that a cache is defined by its total size C_s, its line size L_s, and its associativity K. For simplicity, both C_s and L_s are measured in elements or words of the access we are considering. Two derived parameters that help simplify some expressions are the number of sets in the cache, $N_K = C_s/(KL_s)$, and $C_{sk} = C_s/K$, the cache size devoted to each level of associativity.

Sequential access with uniform probability $\mathbf{Reg_{sp}}(n, p)$

We denote as $AV_{sp}(n, p)$ the cross interference area vector associated to an access $\mathrm{Reg}_{sp}(n, p)$ to n consecutive elements in which each one of them has a probability p of being referenced. The $K + 1$ elements of this vector are calculated as

$$\begin{array}{ll} AV_{sp_i}(n,p) = P(X = K - i) & m < i \leq K \\ AV_{sp_m}(n,p) = P(X \geq K - m) & \\ AV_{sp_i}(n,p) = 0 & 0 \leq i < m \end{array}$$

where $X \in B(n/C_{sk}, 1 - (1 - p)^L_s)$, being $B(n, p)$ the binomial distribution[1] and $m = \max\{0, K - \lceil n/C_{sk} \rceil\}$. The equation is based on the fact that, on average, there are n/C_{sk} lines of the footprint associated to each cache set. Since this is a consecutive memory region, the maximum number of lines a cache set can receive is $\lceil n/C_{sk} \rceil$, so the area vector elements $AV_{sp_i}(n, p)$ for $0 \leq i < m$ must be zero. Also, because of the uniform distribution of the accesses, we know that the number of cache lines per set belongs to a binomial $B(n/C_{sk}, 1-(1-p)^{L_s})$. The probability of

[1] we define the binomial distribution on a non integer number of elements n as $P(X = x), X \in B(n, p) = (P(X = x), X \in B(\lfloor n \rfloor, p))(1 - (n - \lfloor n \rfloor)) + (P(X = x), X \in B(\lceil n \rceil, p))(n - \lfloor n \rfloor)$

access per line of this binomial is easy to calculate, as since each individual element in a cache line has a probability p of begin accessed, and a line holds L_{s} elements, then the probability that at least one of the elements of the line receives a reference is $1-(1-p)^{L_{\mathrm{s}}}$. Since position i, $i>0$, in the area vector represents the ratio of sets that receive $K-i$ lines in the access, its value will be the probability the variable associated to this binomial takes the value $K-i$. The lowest element in the area vector with non-zero probability, m, is the probability the number of lines accessed is $K-m$ or more.

As this is the counterpart of the sequential access described in Section 2.3.2, the autointerference area vector is calculated analogously as $AV_{sp}(C(n)C_{sk})$ being $C(n)$ the average number of lines of the studied vector each line competes with in its associated set, which calculation is described in Section 2.3.2.

Access to groups of elements separated by a constant stride with uniform probability $\mathbf{Reg_{rp}}(N_{\mathrm{r}}, T_{\mathrm{r}}, L_{\mathrm{r}}, p)$

We denote as $AV_{\mathrm{rp}}(N_{\mathrm{r}}, T_{\mathrm{r}}, L_{\mathrm{r}}, p)$ the cross interference area vector associated to an access $\mathrm{Reg}_{\mathrm{rp}}(N_{\mathrm{r}}, T_{\mathrm{r}}, L_{\mathrm{r}}, p)$ to N_{r} regions of T_{r} consecutive elements each and separated by a constant stride of L_{r} elements, in which each individual element has a probability p of being referenced. This area vector is calculated in two phases:

- In a first phase, the region potentially affected by the references is considered. This region allows to measure the impact of the access on the cache by calculating the number of lines that are mapped to each cache set.
- Since accesses really happen with a given probability p, a second phase is needed where the different combinations of accesses are weighted with the probability that they happen.

Calculation of the code footprint We first define the helper function $pos(i) = i \bmod C_{\mathrm{sk}}$, which calculates which position in the cache corresponds to an arbitrary memory position i.

In a first step, the first position C_i of every region i that compounds the pattern mapped on a cache of size C_{sk}, is calculated as

$$\begin{aligned} C_1 &= 0 \\ C_i &= pos(C_{i-1} + L_{\mathrm{r}}), 1 < i \leq N_{\mathrm{r}} \end{aligned}$$

In the following, $CV(i)$ will stand for the number of regions that begin in the position i of the cache. Now we calculate for every cache set, $1 \leq j \leq N_K$, the number of different lines mapped to the considered cache set j in which exactly i of their elements may be referenced by this access pattern. This is the set of values $N(j,i)$, where $1 \leq i \leq L_\mathrm{s}$.

The value of $N(j,i)$ for $i < \min(T_\mathrm{r}, L_\mathrm{s})$ is calculated as

$$N(j,i) = CV(pos(jL_\mathrm{s} - T_\mathrm{r} + i)) + CV(pos(jL_\mathrm{s} + L_\mathrm{s} - i))$$

since only the regions that begin exactly $T_\mathrm{r} - i$ positions before the beginning of the considered set or in the i-th position of the set can contribute with a line where only i of its elements may be referenced by the access pattern.

The calculation of the remaining $N(j,i)$ depends on whether $T_\mathrm{r} < L_\mathrm{s}$. If this is the case, then

$$\begin{array}{ll} N(j,T_\mathrm{r}) = & \sum_{t=0}^{L_\mathrm{s}-T_\mathrm{r}} CV(pos(jL_\mathrm{s}+t)) \\ N(j,i) = & 0, T_\mathrm{r} < i \leq L_\mathrm{s} \end{array}$$

since the regions beginning in the first $L_\mathrm{s} - T_\mathrm{r} + 1$ positions of the set will have one line in which T_r of its elements may be accessed, and given that $T_\mathrm{r} < L_\mathrm{s}$, it is impossible that there are regions with lines where more than T_r elements may be accessed.

Finally, if $T_\mathrm{r} \geq L_\mathrm{s}$, all the $N(j,i)$ but $N(j,L_\mathrm{s})$ have been calculated. The value for the latter is calculated as

$$N(j,L_\mathrm{s}) = \sum_{t=L_\mathrm{s}}^{T_\mathrm{r}} CV(pos(jL_\mathrm{s} - T_\mathrm{r} + t))$$

because any region that begins either in the first position of the set or in the $T_\mathrm{r} - L_\mathrm{s} - 1$ immediately preceding positions will have one line mapped to the considered set j in which all of its elements may be affected by the access pattern.

Weighting the accesses probabilities In the previous phase we have estimated the footprint of this access pattern without taking into account the probability that each element in the footprint is really referenced. Let us remember that the footprint is represented by the values $N(j,i)$, which are the number of lines mapped to set j that contain i words affected by the access pattern. Since the access to each element happens only with probability p, this is an upper bound of the real number of lines

that are accessed. This way, the purpose of this phase is to estimate how many lines are really accessed taking into account that the probability of access to each element in the region is p.

Our strategy to estimate the total area vector for this access pattern is to calculate the area vector for each set j independently and to average them. The area vector for each single set j, S_j, represents the distribution of probability that the access generated references to l different lines mapped to this set for $0 \leq l < K$ in the positions $S_{j(K-l)}$ of the vector, or to K or more different lines, in the position S_{j0}. This distribution of probability is calculated from L_s binomial variables, $X_{ji}, 1 \leq i \leq L_s$, where X_{ji} is the number of lines that are really accessed out of the $N(j,i)$ ones that are mapped to set j and which contain exactly i positions that can be referenced by the access pattern analyzed. This way, $X_{ji} \in B(N(j,i), 1-(1-p)^i)$, where $B(n,p)$ stands for the binomial distribution. The probability of the binomial is given by the fact that if in a given line only i positions may be subject to access, and the access to each position only happens with probability p, then the probability the line has really been accessed is $1-(1-p)^i$. As a result, if we define $X_j = \sum_{i=1}^{L_s} X_{ji}$, then the area vector for the set j can be estimated as $S_{j(K-l)} = P(X_j = l)$, $0 \leq l < K$ and $S_{j0} = P(X_j \geq K)$.

The autointerference is calculated in the same way but the number of lines mapped to each cache set j that contains i words is $max(N(j,i)-1, 0)$ instead of simply $N(j,i)$.

3.3. Condition Dependent PME

In order to consider the probabilities that the different conditional statements that may control a given reference R in its nest hold, we extend the PME that estimates the behavior of a reference R in a loop i with a new argument $\vec{p}$. This vector contains in its position j the probability p_j that the (possible) conditionals that guard the execution of the reference R in nesting level j are verified. If a given loop contains no conditional structures, then $p_j = 1$, which means the execution in this level is unconditional. When there are several nested `IF` statements in the same nesting level, p_j is the product of the probabilities of holding their respective conditions.

We have found that $F_{Ri}(\text{RegIn}, \vec{p})$ may take two different forms when considering codes with data-dependent conditional statements. If the reference is not controlled

by any conditional sentence or if the variable that indexes loop i does not index any of the references found in the condition(s) of the conditional(s) sentence(s) that control the execution of R, then the PME takes the form described in the Equation(2.12) of Section 2.4. This kind of PME disregards its input $\vec{p}$, which is not used in the computations. But if this is not the case, that is, if the variable of the loop is used in the indexing of a data array involved in a conditional that controls the execution of the reference R that is being studied, then a new kind of PME must be used. From now on we will distinguish both kinds of PME by calling the former one *Condition Independent PME* and these new one *Condition Dependent PME*.

Just as we did in Section 2.4, we will now describe the construction of Condition Dependent PME for references that carry no reuse with other references. We will do it in two steps. First, we will develop the general form of a Condition Dependent PME. This PME is based on the probability that the reference that is being analyzed actually accesses each one of the SOLs of the set that the reference can potentially access during one iteration of the loop i we are considering. In a second step, an algorithm to derive this probability will be presented.

General form of a condition dependent PME

A PME must be built for each loop i enclosing a reference R. The PME is basically a summatory where each term is the product of the number of accesses that have a given reuse distance, multiplied by the PME for the lower level when the input footprint corresponds to that reuse distance. When reference R is controlled by data-dependent conditionals, this is, when one or more `IF` statements that depend on the input data control the reference, there is not an unique reuse distance for each line. Depending on the pattern of verification of the conditions that control the execution of the reference, the accesses of `R` may try to reuse SOLs (sets of lines) with very different distances. These reuse distances will have different probabilities of happening, depending on the distribution of probability of the verification of the conditionals that control the execution of the reference. This way, the PMEs for this kind of references will use probabilities not only to represent the miss probability for a given reuse distance, as those in Section 2.4 did, but also to estimate how many accesses take place with each possible reuse distance. Notice that PMEs measure the reuse distance in terms of iterations of the loop they are associated to, and the unit of reuse in a cache is the line. As a result, the base probability to weight the different reuse distances must be the probability that the reference that is being

analyzed accesses one of the SOLs it may potentially access during each iteration of the loop i that is being considered. In general, when the conditionals do not follow an uniform distribution, a set of different probabilities for different iterations and/or SOLs must be used. As the scope of this analysis is restricted to conditionals that follow an uniform distribution, in this work this probability is a single parameter, $Pl_{Ri}(\vec{p})$, that has the same value for every iteration of the loop i and for every SOL that R may access. This way, the condition dependent PME for loop i and reference R has the form

$$F_{Ri}(\mathrm{RegIn}, \vec{p}) = p_i L_{Ri} \sum_{j=1}^{G_{Ri}} \mathrm{WMR}_{Ri}(\mathrm{RegIn}, j, \vec{p}) \ , \tag{3.1}$$

where L_{Ri} is the number of iterations in which new different SOLs would be accessed by reference R due to the stride in loop i if it were not subject to conditional execution. Its calculation is detailed in Section 2.4. p_i is the probability the conditional sentences that control the execution of R in this loop level are true. The product of these two terms gives the average number of iterations in which R accesses different SOLs due to its stride for this loop. This number of iterations must be multiplied by the PME for the immediately lower level evaluated with the appropriate reuse distance area vector, which is what the term WMR_{Ri} stands for, a weighted number of misses for a reference in level i. As stated before, because of the control by data-dependent conditionals, a range of different reuse distances with different probabilities may take place. This range has an average upper bound G_{Ri}, the number of iterations that can potentially reuse the SOLs accessed in the L_{Ri} iterations that give place to accesses to new SOLs. The product of both terms must be equal to the number of iterations of the loop, thus $G_{Ri} = N_i / L_{Ri}$.

Let us now develop the value of $\mathrm{WMR}_{Ri}(\mathrm{RegIn}, j, \vec{p})$, the weighted number of misses generated by reference R in loop i when RegIn is the region accessed since the last access to any of the SOLs affected by the reference of R before loop i begins its execution, and the SOL is used in the j-th possible iteration in which the SOL could be accessed. This function is computed as

$$\begin{aligned} \mathrm{WMR}_{Ri}(\mathrm{RegIn}, j, \vec{p}) = & \overline{Pl_{Ri}(\vec{p})}^{j-1} F_{R(i+1)}(\mathrm{RegIn} \cup \mathrm{Reg}_{Ri}(j-1), \vec{p}) + \\ & \sum_{k=1}^{j-1} Pl_{Ri}(\vec{p}) \overline{Pl_{Ri}(\vec{p})}^{k-1} F_{R(i+1)}(\mathrm{Reg}_{Ri}(k), \vec{p}) \ , \end{aligned} \tag{3.2}$$

where $Pl_{Ri}(\vec{p})$, the probability that R accesses during one iteration of loop i one of

the SOLs that belong to its potential access pattern, is used to weight the probabilities that the different reuse distances take place. In this equation $\overline{p}$ stands for $1-p$, this is, the converse probability of p. Let us remember that $\mathrm{Reg}_{Ri}(n)$ stands for the regions accessed during n iterations of the loop i that may interfere with the accesses of R. The first term in (3.2) considers the case that the SOL has not been accessed during any of the previous $j-1$ iterations. In this case, the RegIn region that could generate interference with the new access to the SOL when the execution of the loop begins, must be added to the regions accessed during these $j-1$ previous iterations of the loop in order to estimate the complete interference region. The references to different data structures often overlap. It is necessary to merge them in only one region in order to avoid having overlapped memory regions considered several times as a source of interference. This addition is performed using the regions union represented by the symbol $\cup\cdot$ The second term weights the probability that the last access took place in each of the $j-1$ previous iterations of the considered loop.

The probability $Pl_{Ri}(\vec{p})$ that reference R accesses one of the SOLs that belong to the region that it can potentially access during one iteration of loop i is a basic parameter to derive $F_{Ri}(\mathrm{RegIn}, \vec{p})$, as we have just seen. This probability depends not only on the access pattern of the reference in this nesting level, but also in the inner ones, so its calculation takes into account all the loops from the i-th down to the one containing the reference. If fact, this probability is calculated recursively in the following way:

- If i is the innermost loop containing R, then

$$Pl_{Ri}(\vec{p}) = \begin{cases} 1 & \text{if the accesses of } R \text{ are consecutive with respect to loop } i \\ p_i & \text{otherwise} \end{cases}$$

  where a consecutive access with respect to a given loop implies that the accesses that take place in consecutive iterations of the loop do reference consecutive memory positions. The condition for this to happen even when the accesses of R depend on an `IF` statement is that the index for the first dimension of R only makes (sequential) progress within the same `IF` statement that controls R.

- If i is not the innermost loop containing R, then

$$Pl_{Ri}(\vec{p}) = \begin{cases} p_i Pl_{R(i+1)}(\vec{p}) & \text{if the index of loop } i+1 \text{ is not used in the} \\ & \text{references found in conditions that control } R \\ p_i \overline{Pl_{R(i+1)}(\vec{p})}^{G_{R_{i+1}}} & \text{otherwise} \end{cases}$$

where we must remember that $\overline{p} = 1 - p$ and that p_i is the product of all the probabilities associated to the conditional sentences controlling R in nesting level i.

Example 10. As an example, we describe now how the equations that model the cache behavior of the reference `B(K,J)` in the code of Figure 3.2 are derived. The innermost loop containing this reference, is also the innermost level. The variable that controls this loop, `J`, is not used in the indexing of referenced found in conditions that control the execution of this reference, thus Equation (2.12) is applied. As this is the innermost loop containing the reference, in the evaluation of this equation, $F_{R3}(\text{RegIn}, \vec{p}) = AV_0(RegIn)$. Since $S_{R2} = N$ and $L_{R2} = H$, the equation for this nesting level is

$$F_{R2}(\text{RegIn}, \vec{p}) = HAV_0(RegIn)$$

The next level is level one. In this level, the loop index indexes references in the two conditional statements that control our reference, so Equation (3.1) applies again. In this case, $S_{R1} = 1$, $L_{R1} = 1 + \lfloor (N-1)/L_s \rfloor$ and $G_{R1} \simeq L_s$, so the equation is

$$F_{R1}(\text{RegIn}, \vec{p}) = p_1 \left(1 + \lfloor (N-1)/L_s \rfloor\right) \sum_{j=1}^{L_s} \text{WMR}_{R1}(\text{RegIn}, j, \vec{p}) \ .$$

When WMR_{R1} is calculated $P_{R1}(\vec{p}) = p_1$

In the outermost level, the variable of the loop indexes a reference in one of the conditions, so we have to apply again Equation (3.1). For this loop and reference, $S_{R0} = 0$, $L_{R0} = 1$ and $G_{R0} = M$, so the equation is

$$F_{R0}(\text{RegIn}, \vec{p}) = p_0 \sum_{j=1}^{M} \text{WMR}_{R0}(\text{RegIn}, j, \vec{p}) \ .$$

In this loop, WMR_{R0} is a function of $P_{R0}(\vec{p}) = 1 - (1 - p_1)^{L_s}$ ■

```
DO I = 1,M
  X = A(I)
  DO J = 1,N
    Y = B(J)
    IF (B(J) .GT. K) THEN
      C(J) = X + Y
    ENDIF
  ENDDO
ENDDO
```

Figure 3.3: Synthetic kernel code

```
posB = 1
DO I = 1, N
  offB(I) = posB
  DO J = 1, M
    IF (A(I,J) .NEQ. 0) THEN
      B(posB) = A(I,J)
      jB(posB) = J
      posB = posB + 1
    ENDIF
  ENDDO
ENDDO
```

Figure 3.4: CRS Storage Algorithm

3.4. Validation

Our validation of the model is based on the comparison of its cache miss predictions with the result of trace-driven simulations. We have used three simple kernels shown in Figures 3.3, 3.4 and 3.2. The code in Figure 3.3 is synthetic kernel with a conditional sentence that control the access to a data structure `C`. Then, Figure 3.4 implements the storage of a matrix in CRS format (Compressed Row Storage), which is widely used to store sparse matrices in a compressed form. The code has two nested loops and a conditional sentence that controls three of the references. Finally, Figure 3.2 is an optimized product of matrices that contains references inside several nested conditional sentences. These conditionals try to avoid unuseful computations when one of their inputs is a zero.

In order to validate our model its predictions were compared with the results of trace drive simulations using different cache configurations, problem sizes and probabilities for the fulfillment of the conditionals for the three example codes. The combinations used to validate the model for each code are shown in Table 3.1. Rows M, N and H correspond to the problem size, this is, the number of iterations of each loop, expressed as the value of its upper limit. Then come the probabilities p_i that the conditional sentences found in the codes are true. The synthetic and the CRS codes have a single conditional and no H loop, thus rows H and p_2 are empty for them. Then, the cache configurations used in the validation are shown in the format $(C_s - L_s - K)$, this is, (cache size-line size-associativity). The cache and line sizes are expressed in bytes. Then, Table 3.1 shows the total number of

Table 3.1: Parameter combinations used for the validation and average and maximum miss rate prediction error

Parameter	Kernel		
	Synthetic	CRS	Matrix Product
M	950,1750,2000, 4500,6000	1000,1200,1400, 1600,1800	350,550, 400,600
N	1200,2500,3000, 4000,9500	1250,1350,2450, 2650,3000	250,350 450,650
H	-	-	600,700,750,800
p_1	0.1,0.2,0.3,0.4,0.5	0.1,0.2,0.3,0.4,0.5	0.1,0.2,0.3,0.4
p_2	-	-	0.1,0.2,0.3,0.4
Cache Configurations ($C_s - L_s - K$)	32K-32-1	32K-32-1	32K-32-1
	32K-32-2	32K-32-2	32K-32-2
	64K-32-1	64K-32-1	-
	64K-32-2	64K-32-2	64K-32-2
	128K-64-2	128K-64-2	128K-64-2
Combinations	625	625	4096
Avg Δ_{MR}	0.22%	1.43%	2.23%
Max Δ_{MR}	3.81%	8.05%	11.32%

parameter combinations tried for each code taking into account the previous rows. For each one of these combinations a total of 25 different simulations were made using different base addresses for the data structures. This improves the validation of the model by taking into account many different relative positions for the mapping on the cache of the different data structures. The last two rows in the table show the average and the maximum value for each code of the metric Δ_{MR} that we use to measure the accuracy of the model. This metric is the average of the absolute value of the difference between the predicted and the measured miss rate (MR) expressed as a percentage in each one of the 25 simulations performed for each parameter combination. As expected, the average and maximum errors grow with the complexity of the code. Still, we consider that a maximum absolute error of only about 11% is very satisfactory. Also, the large difference between the average and the maximum Δ_{MR} shows that (relatively) large errors are very infrequent and, in general, the predictions estimate well the cache behavior.

Tables 3.2, 3.3 and 3.4 show the validation results for some randomly chosen combinations of the problem size, the conditional probabilities and the cache configurations for the three codes proposed in Figs. 3.3, 3.4 and 3.2, respectively. The

Table 3.2: Validation data for the synthetic kernel in Fig. 3.3 for several cache configurations, problem sizes and condition probabilities

M	N	p	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
50000	47500	0.4	128K	64	2	0.015	182.211	68.022	0.005
50000	47500	0.2	64K	256	4	0.004	138.187	50.003	0.005
22000	14500	0.4	256K	128	4	0.001	28.244	7.033	0.003
22000	14500	0.4	64K	64	1	0.067	65.002	7.129	0.004
18000	22000	0.2	256K	128	2	0.574	23.021	7.586	0.004
18000	22000	0.1	128K	64	2	0.076	22.112	6.012	0.004
18000	22000	0.3	32K	256	4	0.141	95.223	8.010	0.004
14500	19500	0.7	128K	64	8	0.000	32.224	7.697	0.005
14500	19500	0.2	128K	32	2	0.252	20.269	5.331	0.005
14500	19500	0.3	64K	32	1	0.124	20.901	6.465	0.004
1750	1750	0.4	64K	4	64	0.000	1.123	1.000	0.003
1750	1750	0.7	64K	8	32	0.000	0.988	0.322	0.003

columns in the three tables have the same meaning as the respective rows in Table 3.1. Many of the combinations chosen in these tables do not belong to the set of experiments described by Table 3.1, so that the behavior of the model can be analyzed for a wider scope of parameters. The last three columns in each table correspond, respectively, to the simulation time, execution time and modeling times expressed in seconds and measured in a Athlon 2400 processor-based system (2,086 GHz). As we see, modeling times are much shorter than trace-driven simulation times despite the fact that we use a very fast and simple simulator. In fact, many times they are even faster than the native execution times. Furthermore, sometimes modeling times are several orders of magnitude shorter than trace-driven simulation and even execution times. The modeling time does not include the time required to build the equations for the example codes as the equations are developed by hand. The time necessary to execute the model is always less than 1 second.

Figures 3.5 and 3.6 show the evolution of both the number of misses and the miss rate measured and predicted for different cache configurations and probabilities of the conditionals for the CRS and the matrix product codes, respectively. The figures show, as the previous tables, that the model is successful in predicting the behavior of the cache. A new interesting conclusion we can draw from these figures is that our extended model is indeed required to predict correctly the behavior of the memory hierarchy when irregular access patterns are involved. We can see that a simplified

Table 3.3: Validation data for the CRS code in Fig. 3.4 for several cache configurations, problem sizes and condition probabilities

M	N	p	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
6200	10150	0.4	256K	64	4	0.01	16.308	4.022	1.225
4200	17150	0.1	32K	32	2	0.04	14.797	6.401	0.246
16220	7200	0.2	128K	32	2	0.03	27.477	5.011	3.646
6200	14250	0.3	512K	64	4	0.00	21.089	5.891	1.221
9200	14250	0.1	32K	32	8	0.04	37.768	11.001	1.196
1100	15550	0.5	32K	32	8	0.02	2.724	1.668	0.021
2900	17250	0.3	256K	128	4	0.17	10.363	4.573	0.572
8900	9250	0.1	256K	64	4	0.64	17.119	11.228	2.516
4200	12150	0.1	32K	32	2	0.04	9.364	3.880	0.246
5000	15000	0.3	256K	64	4	0.11	17.852	10.330	0.804
7200	12250	0.1	32K	32	8	0.04	18.224	9.646	0.721

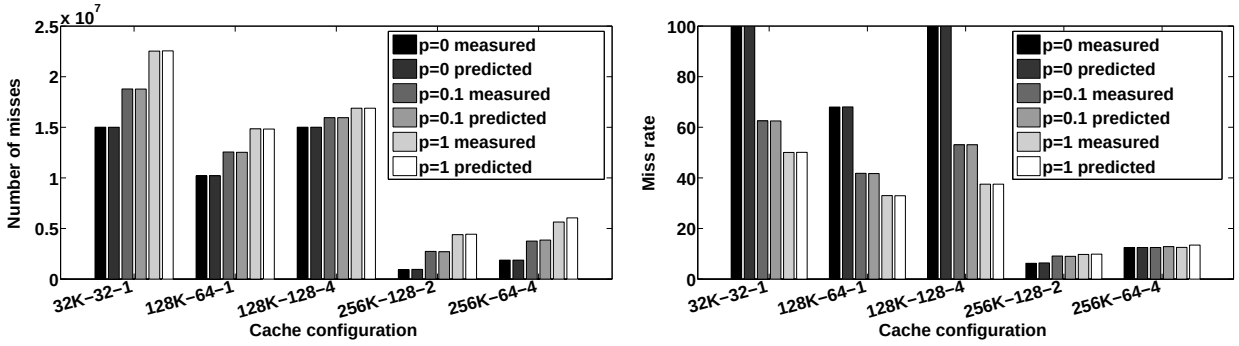
model that did not support irregular access patterns and which chose to make all probabilities either 0 or 1 (the two extremes cases) would yield predictions very different from the real values obtained for intermediate probabilities like 0.1, shown in the figures. This justifies the interest of our research.

Figure 3.7 compares the miss rate measured and the miss rate predicted for the CRS storage and matrix product codes when the probability of verification of the condition takes different values between 0.1 and 0.9. The accuracy of the prediction is good in all the situations, while the miss rate of the code is highly dependent on the probability of the conditions in the code. The miss rate is higher when the probability of verification is lower because accesses are much more irregular. This way, it is important to feed the model with the right values of the probabilities of the conditional statements because the miss rate can be mispredicted otherwise.

Finally, Figure 3.8 contains the evolution of Δ_{MR} for the CRS storage and the optimized matrix product respectively for different cache configurations and matrix sizes. The prediction is more accurate in the CRS storage code, but it is still good for the product of matrices.

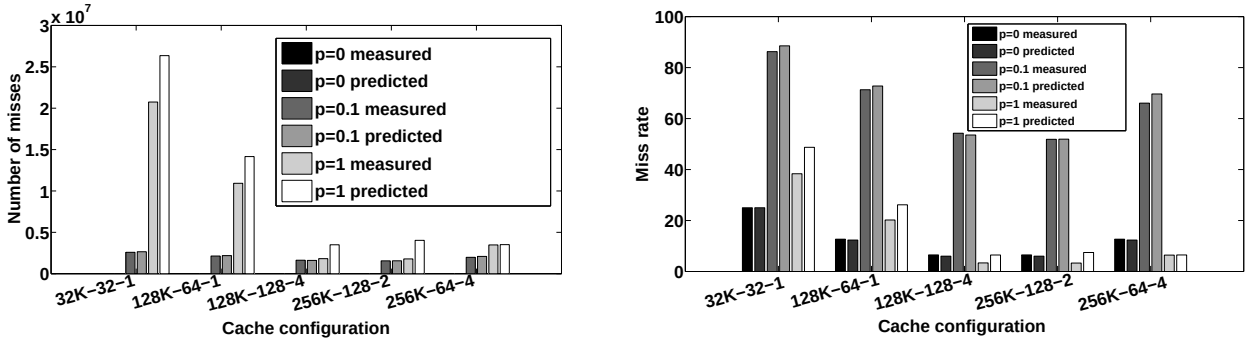
Table 3.4: Validation data for the optimized matrix product code in Fig. 3.2 for several cache configurations, problem sizes and condition probabilities

M	N	H	p_1	p_2	C_s	L_s	K	Δ_{MR}	T_sim	T_exe	T_mod
750	750	1000	0.2	0.1	128K	64	8	0.79	24.444	11.233	0.203
750	750	1000	0.8	0.3	128K	128	16	1.31	86.845	72.069	0.987
900	850	900	0.9	0.1	512K	64	8	0.59	85.358	65.266	0.990
900	950	1500	0.1	0.4	256K	64	4	6.62	31.768	16.201	0.511
900	950	1500	0.8	0.3	128K	32	2	2.04	171.755	85.023	0.149
1000	850	900	0.7	0.5	32K	64	2	3.13	110.328	108.211	0.139
200	250	150	0.8	0.2	128K	32	2	0.48	0.764	0.550	1.034
200	250	150	0.1	0.3	256K	64	4	5.91	0.134	0.112	0.301
200	250	150	0.3	0.1	32K	32	8	1.45	0.406	0.323	0.030
100	350	90	0.8	0.5	32K	32	8	0.14	0.500	0.201	0.031
100	350	90	0.4	0.4	64K	64	4	0.40	0.218	0.122	0.586
100	350	90	0.2	0.3	32K	64	2	0.05	0.104	0.101	0.309



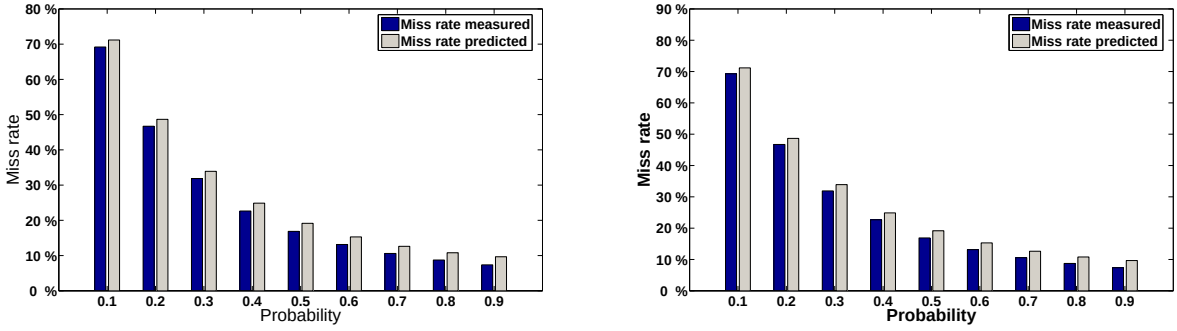
(a) Number of misses (b) Miss rate

Figure 3.5: Measured versus predicted (a) misses and (b) miss rates for several cache configurations and different probabilities of verification of the conditionals for the CRS code (see Figure 3.4) with $M = 1500$ and $N = 10000$. The cache configurations are expressed as (Cs-Ls-K), with sizes in bytes.



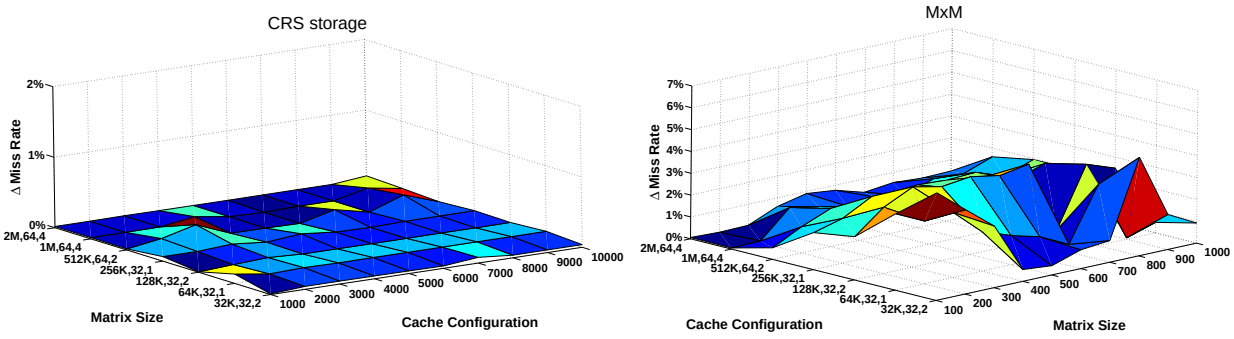
(a) Number of misses (b) Miss rate

Figure 3.6: Measured versus predicted (a) misses and (b) miss rates for several cache configurations and different probabilities of verification of the conditionals for the optimized matrix product code (see Figure 3.2) with $M = 300$, $N = 300$ and $H = 300$. The cache configurations are expressed as (Cs-Ls-K), with sizes in bytes.



(a) (b)

Figure 3.7: Measured versus predicted miss rates for different probabilities of verification of the conditionals for the CRS storage code and the optimized matrix product a 2-way cache of 512 KBytes with 64 bytes per cache line. The matrix sizes were $M = N = 10000$ in the CRS storage code and $M = N = H = 1000$ in the optimized product of matrices.



(a) (b)

Figure 3.8: Surfaces representing the Δ_{MR} evolution for different cache configurations and matrices sizes in the CRS storage and optimized matrix product codes. The cache configuration is denoted using the notation (C_{s},L_{s},k).

Chapter 4

Model Extension to Handle Codes with Indirections

In the previous chapter we proposed an extension to the PME model to consider codes with irregular access patterns due to conditional statements. They constitute an important subset of the codes with irregular access patterns.

Another important source of irregularity is the existence of indirections, that is, references where the indexing of an array, called the base array, is done across the values contained in another array known as the index array. This array can be also accessed using an indirection, so, more than one level of indirection is possible. For example, the management of compressed matrix storage methods used in sparse computation gives place to a big numer of indirections.

Our model considers indirections in which all the elements of the array accessed by means of the indirection have the same probability of being accessed, i.e., where the irregular access is uniformly distributed on the referenced array. In sparse computation this implies that the nonzeros should be uniformly distributed along the sparse matrix. This restriction eases the treatment of the problem in this first attempt to model automatically the cache behavior of codes with indirections, while allowing to represent the most important problems that irregular access patterns pose for their modeling. The model is also extended to cover an important class of non-uniform irregular access patterns. Namely, we consider the indirections generated by the compressed storage of realistic banded matrices, a very common distribution in sparse matrices [27].

```
DO I0 =1, N0
  DO I1 =1, N1
  ...
    DO IZ =1, NZ
     ...
      A(fA1(IA1), ..., fAj(B(fB1(IB1))), ...)
     ...
    END DO
  ...
  END DO
END DO
```

Figure 4.1: Nested loops with structures accessed using indirections.

Section 4.1 contains a description of the extended scope of application of the PME. Separated extensions are proposed for the modeling of codes with indirections when the data involved follows an uniform distribution and when banded matrices are managed. Section 4.2 is devoted to the model extension for uniform distributions while Section 4.3 covers the treatment of banded matrices.

4.1. Scope of Application

Figure 4.1 depicts the scope of application of our extended model. It shows a set of normalized perfectly or non-perfectly nested loops in which the number of iterations of every loop must be the same in every execution of the loop. The reference indexes are affine functions f_i either of the loops control variables $\mathtt{I}_i$ or of values read from arrays. We call index or indirection array the one whose values are used to index another array, which we call the base array of the indirection. Index arrays can be themselves indexed by other arrays, which gives place to several levels of indirection.

4.2. Model Extension for Uniform Distributions

In the first extension proposed in this chapter, the probability that a component of the base array of an indirection is accessed is uniform. This means they all have the same probability of being accessed. Some adaptations must be done to the model

```
function Reg_Ri(n) {
    RegSet = ∅
    foreach array A involved in the code {
        R_A = ∅
        foreach reference R' to array A in loop i or loops nested inside it {
            R_A = merge(region_accessed(R, i, n), Q_R, R_A)
        }
        reg_A.region_func = access_pattern(R_A)
        reg_A.is_self_interference = (A == array(R))
        RegSet = RegSet ∪ reg_A
    }
    return RegSet
}
```

Figure 4.2: Calculation of $Reg_{Ri}(n)$, the set of regions that can interfere with the attempts of reuse of reference R generated during n iterations of the loop at nesting level i.

to cover this new situation. The miss probability estimation process is adapted in Section 4.2.1 to cover the access pattern recognition of codes with indirections. New equations are proposed in Section 4.2.2 for codes with indirections with an uniform distribution. In Section 4.2.3 a small modification to model codes involving banded matrices with a uniform distribution of the values inside the band, is proposed. Section 4.2.4 shows the main validation results achieved with this model extension.

4.2.1. Miss Probability Estimation in Codes with Indirections

As we established in Section 2.3, the miss probability estimation is a process that can be divided in three steps : the access pattern identification, the cache impact quantification and the area vectors union. In the previous chapter, some adaptations were proposed to the two first steps to cover the modeling of irregular codes due to conditional statements. In the access pattern identification we introduced some changes to identify correctly the irregular access pattern due to conditional statements. We also introduced in the cache impact quantification step two new types of irregular regions that allowed us to measure the cache impact of irregular access patterns. Finally, the area vectors union did not suffer any change. Figure 4.2 shows the pseudocode for the calculation of the interference region of reference R during the execution of n iterations of nesting level i: the access pattern of the references to each array A found within the loop is identified in turn and added to the set of

```
function region_accessed(R, h, n) {
    A = array(R)
    foreach dimension j of A {
        i = loop_that_indexes_dimension(R, j)
        if not indexed_by_indirection(R, j) {
            R_Rj(h, n) = (iters_regular(i, h, n), S_Ri, 1)
        } else {
            R_Rj(h, n) = (iters_irregular(i, h, n), S_Ri,
                                        prob(R, i, h, n))
        }
    }
    return simplify(R_R(h, n))
}
```

(a) Calculation of $\mathcal{R}_R(h,n)$, the tuple-based representation of the region accessed by reference R during n iterations of the loop at nesting level h.

```
function access_pattern(R) {
    if R == (M, S, P) {
        if P == 1.0
            if S == 1
                return Reg_s(M)
            else
                return Reg_r(M, 1, S)
        else
            if S == 1
                return Reg_sp(M, P)
            else
                return Reg_rp(M, 1, S, P)
    } else {
        if R == ((M_1, 1, P_1), (M_2, S_2, P_2))
            return Reg_rp(M_2, M_1, S_2, P_1 · P_2)
        else
            return error(UNKOWN)
    }
}
```

(b) Identification of the access pattern associated to the memory region described by the tuple(s) $\mathcal{R}$.

Figure 4.3: Identification of the access pattern followed by the references during a reuse distance.

regions accessed. The memory regions associated to the same array R accesses are marked because its cache impact quantification step is different.

In this extension for codes with indirections the last two steps of this process remain the same. The irregular access patterns followed by the references due to the existence of indirections are the same as those ones identified in codes with conditional statements. However, the access pattern identification step needs some modifications to identify correctly the irregular access patterns due to indirections.

Access Pattern Identification

Figure 4.3 shows the two steps involved in the identification of the access pattern that references follow during a reuse distance consisting of n iterations of the loop at nesting level h. Function *region_accessed* analyzes the indexes of the studied reference R to obtain a numerical representation of its access pattern during the considered period of the execution of the code. In function *access_pattern*, this representation is mapped to a given access pattern.

In function *region_accessed*, for each reference R the indexes of each dimension and the number of iterations of each loop during this reuse distance are examined. The output of this analysis is a D_{A}-tuple $\mathcal{R}_R(h,n)$, as in the case of regular codes described in Section 2.3.1, where D_{A} is the number of dimensions of the array `A` referenced by R. In regular codes each element of this tuple consisted in its turn of a 2-tuple $\mathcal{R}_{Rj}(h,n) = (M_j, S_j)$ where the M_j is the number of different points accessed along dimension j and S_j the constant stride between two consecutive points. In codes with indirections this tuple has a third component P_j that stands for the probability each one of these points is actually accessed by R. If the access is not indexed using an indirection $P_j = 1$. Function *access_pattern* uses the information contained in this tuple to determine the access pattern followed by that reference.

The algorithm followed to calculate the 3-tuple associated to dimension j of reference R during n iterations of the loop at nesting level h is described now. When the indexing of dimension j is not done across an indirection then the method used to calculate this tuple is the one described in Section 2.3.1 for the regular case with $P = 1$. But when the indexing of dimension j depends on an indirection, that is, when the index has a form α_{Rj}·`B(`$f($`I`$_i)$`)`$+\delta_{Rj}$, we assume that the accesses are spread uniformly on the dimension j of the array. Since our indirection is multiplied by some constant α_{Rj} (usually one), there are $\lfloor D_{\mathrm{A}j}/\alpha_{Rj} \rfloor$ different points in the dimension that can be actually accessed (e.g. reference `A(2*B(I))` can only access the even elements of array `A`), where $D_{\mathrm{A}j}$ is the number of elements in dimension j. Each point has an uniform probability $1/\lfloor D_{\mathrm{A}j}/\alpha_{Rj} \rfloor$ of being the one accessed because of each given value read from the index array. As a result, if $\mathrm{Iters}_i(h,n)$ (see Section 2.3.1) different values have been read from the index array `B` during n iterations of the loop at nesting level h, where i is the nesting level of the loop whose index controls the accesses to the index array `B`, the average probability each that each one of the points that R can access in the j-th dimension of its base array has been accessed at least once is $1 - (1 - 1/\lfloor D_{\mathrm{A}j}/\alpha_{Rj} \rfloor)^{\mathrm{Iters}_i(h,n)}$, thus

$$\mathcal{R}_{Rj}(h,n) = \left(\left\lfloor \frac{D_{\mathrm{A}j}}{\alpha_{Rj}} \right\rfloor, S_{Ri}, 1 - \left(1 - \frac{1}{\lfloor D_{\mathrm{A}j}/\alpha_{Rj} \rfloor} \right)^{\mathrm{Iters}_i(h,n)} \right) \tag{4.1}$$

Once $\mathcal{R}_R(h,n)$ has been calculated for the reference that reads the indexing array `B`, it is straightforward that the number of different points the reference accesses is $\prod_{k=1}^{D_{\mathrm{B}}} M_k P_k$, i.e, the product of the number of different points it may access in each dimension multiplied by the probability each one of such accesses actually takes

place.

As in Section 2.3.1, there are possible simplifications in between pairs of 3-tuples $\mathcal{R}_{Rj}(h,n)$ that describe the access pattern in different dimensions of the array:

$$\begin{array}{lcl} ((1,S_j,P_j),(M_k,S_k,P_k)) & = & (M_k,S_k,P_j \cdot P_k) \\ ((M_j,S_j,P_j),(M_k,M_j \cdot S_j,P_k)) & = & (M_j \cdot M_k,S_j,P_j \cdot P_k) \end{array}$$

After these simplifications a single 3-tuple (M_s,S_s,P_s) that describes the region accessed by the reference is typically obtained.

The notation described above suffices for the representation of memory regions in codes in which there is a single reference per data structure. In codes in which several references access the same data structure, the regions they access will often overlap or be adjacent, so we have developed simple algorithms to merge the descriptors for overlapping or adjacent regions. This way, lines that are accessed by different references are not taken into account several times as source of interferences. In order to perform this merging, one more parameter is used to describe the region affected by a given reference R: the position Q_R with respect to the beginning of the array of the first element it contains. The merging algorithm is applied in function *merge* in Figure 4.2 and it is described in [31].

As Section 2.3 explains, rather than this description of the memory region accessed, the output of the access pattern identification step is a function that characterizes the access pattern whose output is the area vector associated to it. Depending on the values of S_s and P_s in a tuple $\mathcal{R}_{Rj}(h,n)$, four kinds of access pattern functions can be identified (see Figure 4.3(b)):

1. When $P_s = 1$, it is a regular access pattern so the process described in Section 2.3.1 is followed, based on M_s and S_s.

2. When $P_s < 1$ the access pattern is irregular, as each point involved in the pattern has only a certain probability of being actually accessed:

   a) If $S_s = 1$, it is an access to M_s consecutive elements in which each element is accessed with a probability P_s. The function that calculates the area vector for this access is $\mathrm{Reg}_{\mathrm{sp}}(M_s,P_s)$.

   b) Otherwise the access affects M_s different points separated by a constant stride S_s, which each element is accessed with a probability P_s. The

```
DO I=1,M
  REG=0
  DO J=R(I), R(I+1) - 1
    REG = REG + A(J) * X(C(J))
  ENDDO
  D(I)=REG
ENDDO
```

Figure 4.4: Sparse Matrix-Vector Product

area vector associated to this access pattern is estimated by function $\mathrm{Reg}_{\mathrm{rp}}(M_s, 1, S_s, P_s)$.

Sometimes it is not possible to reduce $\mathcal{R}_R(h, n)$ to a single tuple. All the cases of this kind we have found in the codes we have analyzed had the form $\mathcal{R}_R(h, n) = ((M_1, 1, P_1), (M_2, S_2, P_2))$, which can be represented by the function $\mathrm{Reg}_{\mathrm{rp}}(M_2, M_1, S_2, P_1 \cdot P_2)$, as they are an access to M_2 separate groups of M_1 consecutive elements each that are separated by a constant stride S_2, having each individual element of the region a probability $P_1 \cdot P_2$ of being accessed.

Example 11. Let us consider the code of Figure 4.4, which is part of the Sparskit toolkit [43]. This code performs the product of an sparse matrix stored in CRS format and a vector. The CRS format stores sparse matrices by rows in a compressed way using three vectors. One vector stores the nonzeros of the sparse matrix ordered by rows, another vector stores the column indexes of the corresponding nonzeros, and finally another vector stores the position in the other two vectors in which the data of the nonzeros of each row begins. In our codes we always call these vector `A`, `C` and `R` respectively. The codes which manipulate compressed sparse matrices contain many indirections. We will illustrate the extended access pattern identification step for codes that contain indirections identifying the memory regions accessed during one iteration of the outermost loop (loop `I`) of this code.

In each iteration of loop `I` a whole execution of the loop `J` takes place. The average number of iterations of this loop is N_1. Since it sweeps along the elements of a row of the sparse matrix, its value is $N_1 = Nnz/M$, where Nnz is the number of nonzeros in the sparse matrix and M its number of rows, let's remember we assume an uniform distribution of the nonzeros. The number of nonzeros can be assumed from the size declared for the arrays `A` and `C`, or be part of a directive to the compiler or be extracted from a profiling of the input data. Reference `A(J)` is indexed by the

variable that controls this loop, so it sweeps along N_1 different elements with stride 1 with probability one. Thus, its $\mathcal{R}_{R1}(0,1) = (N_1, 1, 1)$, whose area vector can be estimated by $\mathrm{Reg_s}(N_1)$

Reference `C(J)` follows exactly the same access pattern, thus it also accesses a region $(N_1, 1, 1)$ whose associated area vector is estimated by $\mathrm{Reg_s}(N_1)$.

Reference `X(C(J))` is indexed by the variable of the loop indirectly, through a read from vector `C`. This way, applying Equation (4.1), its $\mathcal{R}_{R1}(0,1)$ is estimated as $(D_{\mathrm{X1}}, 1, 1 - (1 - 1/D_{\mathrm{X1}})^{N_1})$. The simplest function that can estimate the area vector for this access pattern is $\mathrm{Reg_{sp}}(D_{\mathrm{X1}}, 1 - (1 - 1/D_{\mathrm{X1}})^{N_1})$, where D_{X1} is the first dimension of array `X`.

Also, during one iteration of loop `I`, reference `D(I)` accesses a single element of vector `D`, thus its $\mathcal{R}_{R1}(0,1) = (1, 1, 1)$, whose area vector is given by $\mathrm{Reg_s}(1)$ ■

4.2.2. PMEs for Codes with Indirections

The PME model for regular codes has an unique equation that can characterize the cache behavior of any reference with a regular access pattern. In the previous chapter, an additional equation was proposed to cover the modeling of references guarded by one or more conditional statements. These two equations allow to model codes with irregular access patterns due to data-dependent conditional statements.

In the extension of the PME model to cover codes with indirections, the construction of F_{Ri} depends on whether the control variable for loop i, $\mathtt{I}_i$, is used in the indexes of index arrays found in the reference or not.

If $\mathtt{I}_i$ does not appear in R, or if it only appears in the indexes that do not depend on indirections, i.e., indexes of the form $\alpha\mathtt{I}_i + \delta$, the access pattern of R is regular with respect to loop i, so the PME for this loop is built using the Equation 2.12 in Section 2.4.

When the control variable for loop i, $\mathtt{I}_i$, indexes an index array in an indirection, the access pattern on the base array of our reference R is irregular with respect to loop i. The reason is that the position accessed by R no longer depends directly on $\mathtt{I}_i$, but on the value read from the array that $\mathtt{I}_i$ indexes either directly or through more levels of indirection.

The distribution of the values read from the index arrays on the dimension of

the base array they index determines the accesses, the reuses, and thus the PME that models the reference-cache interaction. In our modeling we assume that this distribution is uniform, that is, all the elements of the base array have the same probability of being accessed in each iteration of the considered loop.

We have found that two classes of irregular access patterns arise depending on whether the values of the considered index array are ordered or not. Monotonic irregular access PMEs model the situation when the accesses generated by the indirection are ordered, i.e., when the values read from the index array are monotonically increasing or decreasing. When this condition does not hold or we simply do not have information about the indexing values, non-monotonic irregular access PMEs are going to be applied. We now explain the two kinds of PMEs in turn.

Monotonic Irregular Access PME

When they are ordered, the sequence of accesses produced by the indirection can be characterized as a monotonically increasing or decreasing function. In this case, the reuses in the considered loop i can only take place with respect to the line referenced in the immediately previous iteration. This way, the PME for regular access patterns explained in Section 2.4 (Equation (2.12)) can be used in this situation, the difference being that L_{Ri}, the number of iterations of this loop that cannot exploit reuse, or conversely, the number of different sets of lines (SOLs) that R accesses during the execution of the loop, cannot be estimated as in the regular access pattern case. In a monotonic irregular access pattern,

$$L_{Ri} = D_{Ri}(1 - (1 - L_{Ri_1}/D_{Ri})^{N_i}) \tag{4.2}$$

where D_{Ri} is the number of different SOLs that R can potentially access during the execution of the loop i and L_{Ri_1} is the number of SOLs accessed during one iteration of loop i. The rational for Equation (4.2) is that if in each iteration of the loop i, on average L_{Ri_1} different SOLs are accessed out of the D_{Ri} ones that R could access, then each one of them has the same uniform probability $p_{Ri} = L_{Ri_1}/D_{Ri}$ of being accessed in each iteration of the loop. Thus, the probability that a SOL has been accessed at least once during the N_i iterations of the loop is $1 - (1 - p_{Ri})^{N_i}$. Multiplying this probability by the number of SOLs yields the average number L_{Ri} of different SOLs that are actually referenced. So this is the number of iteration of the loop in which no reuse is possible. Because the values in the index array are monotonically increasing (or decreasing), the other $N_i - L_{Ri}$ iterations of the loop

attempt to reuse the SOL accessed in the immediately previous iteration, with a reuse distance of one iteration of the loop, as PME (2.12) reflects.

The number L_{Ri_1} of different SOLs accessed during one iteration of loop i is trivially one in the innermost loop z that contains R. For any other loop i, L_{Ri_1} is L_{Rk}, with $k = \min\{v/i < v \leq z \wedge \text{DimInd}(v) = \text{DimInd}(i)\}$, i.e., it is the L_R for the outermost loop k nested inside loop i such that its index variable $\texttt{I}_k$ indexes (indirectly) the same dimension of the base array `A` referenced by our reference R as the variable $\texttt{I}_i$ of the considered loop i. If no such loop exists, then, again, $L_{Ri_1} = 1$. Another way to express it is that the number of different SOLs accessed in one iteration of loop i is the number of SOLs accessed during the complete execution of the outermost loop nested inside loop i that indexes, indirectly, the same dimension of the affected base array as $\texttt{I}_i$. This definition allows to handle correctly those cases in which, for example, the indirection for a given dimension in R depends on several loop index variables, e. g.: in `A(B(I,J))` both `I` and `J` index indirectly the only dimension of vector `A`. Another example for this situation is often found in the codes in which indirections are generated by sparse matrices because of the formats used to store them.

Example 12. If we analyze the sparse matrix-vector product code in Figure 4.4, we see that vector `X` is accessed indirectly through `C(J)` in the innermost loop, whose index variable is precisely `J`. In that loop, trivially, $L_{R1_1} = 1$ for reference `X(C(J))`. If we analyze the outer loop on `I`, we can see that this variable indexes `R(I)`, which defines the values for `J`. As as result the indexes of both loops index indirectly the only dimension of vector `X`, and thus, for the outermost loop 0, $L_{R0_1} = L_{R1}$. ■

We complete our modeling for this access pattern with the expression of D_{Ri} :

$$D_{Ri} = \left\lceil \frac{D_{\mathrm{A}j} d_{\mathrm{A}j}}{\max\{S_{Ri}, L_s\}} \right\rceil \tag{4.3}$$

where $S_{Ri} = \alpha_{Rj} \cdot d_{\mathrm{A}j}$ and $d_{\mathrm{A}j}$ are defined as in the preceding section, and $D_{\mathrm{A}j}$ is the size or number of elements along the j-th dimension of the array `A` referenced by R. Let us remember that j is the dimension that is indexed, in this case indirectly, by $\texttt{I}_i$. This also means that in this case the constant α_{Rj} is multiplying the indirection indexed by $\texttt{I}_i$ rather than the variable $\texttt{I}_i$ itself.

Example 13. From the shape of the loops displayed in Figure 4.4 a compiler can speculate that `R` stores the indices for the beginning of the data of each row of

the sparse matrix in `A` and `C`, which hold the nonzeros and their corresponding columns, respectively. Another possibility to extract this information would be to include a directive to the compiler in the code reporting which is the role of each array in the storage of the sparse matrix. With this knowledge we can also infer that the sparse matrix has M rows and we can speculate that the values in `C` are ordered for each row. If this were the case we could conclude that the values read in Figure 4.4 by `C(J)` are monotonically increasing during each whole execution of the loop `J`, at nesting level 1. As a result, the access pattern of `X(C(J))` in this loop can be modeled by a monotonic irregular access PME. This PME has the form of Equation (2.12), with its L_{Ri} calculated according to Equation (4.2). The latter expression is a function of, D_{R1}, the number of different SOLs that R can potentially access during the execution of the loop `J`, and L_{R1_1}, the number of SOLs accessed during each iteration of this loop.

Equation (4.3) allows to calculate D_{R1} knowing that (a) the indirection takes place in the first dimension of the base array `X` ($j = 1$), (b) the cumulative size for the first dimension of any array is always one ($d_{\mathrm{X1}} = 1$), (c) the stride S_{R1} of our reference with respect to its indirection is one ($S_{R1} = \alpha_{R1} \cdot d_{\mathrm{X1}} = 1 \cdot 1$), and (d) the size of the first (and only) dimension of `X` is a value D_{X1} our compiler extracts from the definition of the vector in the code. With these data we evaluate Equation (4.3) as $D_{R1} = \lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$. This means that during each iteration of the loop `J`, `X(C(J))` could potentially access any of the $\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$ lines that constitute `X`.

Both in our general explanation about the calculation of L_{Ri_1} and in our preceding example, we explained that trivially, in the innermost loop that contains a reference R with an indirection, $L_{Ri_1} = 1$, which is the case for `X(C(J))` in loop `J`.

With these two pieces of data we can evaluate Equation (4.2):

$$L_{R1} = \lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil \left(1 - \left(1 - \frac{1}{\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil} \right)^{N_1} \right) . \tag{4.4}$$

This expression assumes that each one of the $\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$ lines of `X` has the same uniform probability of being accessed during each one of the N_1 iterations of loop `J`. As a result, after the N_1 iterations, each line has a probability $1-(1-1/\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil)^{N_1}$ of having been accessed at least once. Thus multiplying this probability by the number of lines we get the number of different lines that were actually accessed on average. As for the average number of iterations of this loop N_1, since it sweeps along the elements of a row of the sparse matrix, its value is $N_i = Nnz/M$, where

Nnz is the number of nonzeros in the sparse matrix and M its number of rows. The number of nonzeros can be assumed from the size declared for the arrays `A` and `C`, or be part of a directive to the compiler or be extracted from a profiling of the input data.

Once we have calculated the number L_{R1} of different SOLs we access in each execution of the loop (with each SOL consisting of a single line in this case), we can replace it in Equation (2.12). This equation will consider the L_{R1} first accesses to a different line with a miss probability that depends on reuses that take place with respect to accesses outside the loop, while the remaining $N_1 - L_{R1}$ accesses necessarily try to reuse the line accessed in the immediately preceding iteration. As a result, the miss probability for them is associated to the regions accessed during one iteration if this loop ■

Non-Monotonic Irregular Access PME

When the indexing values are not monotonic, or we have no information about their ordering, the last access of a reference to a given line, or in general, set of lines (SOL), in the considered nesting level i may have happened an indeterminate number of iterations ago. The number of loop iterations between two accesses of the reference to the same SOL is not a fixed value, since every SOL can be accessed with a given probability in each iteration of the loop. Thus, a probabilistic approach must be followed to estimate the number of misses taking into account that each potential reuse distance happens now with a different probability.

In the presence of uniform probabilities, each one of the D_{Ri} different SOLs that R can potentially access during each execution of the loop has the same probability $p_{Ri} = L_{Ri_1}/D_{Ri}$ of being accessed in a given iteration, no matter the accesses are monotonic or not. Also, every SOL has this probability of access in each one of the N_i iterations of the loop. As a result, the number of misses generated by a non-monotonic irregular access pattern during the execution of loop i can be estimated by means in a summatory in which each term estimates the number of misses that the accesses of R can generate in the j-th iteration of the loop:

$$F_{Ri}(\text{RegIn}) = \sum_{j=1}^{N_i} WM_{Ri}(\text{RegIn}, j), \tag{4.5}$$

where $WM_{Ri}(\text{RegIn}, j)$ yields the weighted number of misses generated in the j-th

potential access of R to the SOLs it defines in loop i. In this expression, RegIn stands for the region accessed since the last reference to the SOLs that R accesses in this loop when the execution of this loop begins, as usual. This number of misses is calculated as

$$\begin{aligned} WM_{Ri}(\text{RegIn}, j) =& (1-p_{Ri})^{j-1} \cdot F_{R(i+1)}(\text{RegIn} \cup \text{Reg}_{Ri}(j-1)) + \\ & \sum_{h=1}^{j-1} p_{Ri} \cdot (1-p_{Ri})^{h-1} \cdot F_{R(i+1)}(\text{Reg}_{Ri}(h)) \ , \end{aligned} \tag{4.6}$$

where $p_{Ri} = L_{Ri_1}/D_{Ri}$, as explained in the previous section, yields the probability that a given SOL of the base array that R can potentially access during the execution of loop i is indeed accessed during one iteration of that loop.

The first term in (4.6) considers the case that the SOL has not been accessed in any of the previous $j-1$ iterations, which is $(1-p_{Ri})^{j-1}$ given that p_{Ri} is the probability of access in each iteration. In this case, the RegIn region that could generate interference with the new access to the line when the execution of the loop begins must be added to the $\text{Reg}_{Ri}(j-1)$ regions accessed during these $j-1$ previous iterations of the loop in order to account for the complete interference region. This addition is represented by means of the $\cup$ operator. The second term weights the probability that the last access took place in each one of the $j-1$ previous iterations of loop i. The probability that the last access to a given SOL was exactly h iterations before the current iteration is $p_{Ri} \cdot (1-p_{Ri})^{h-1}$, that is, the probability there was an access to the SOL h iterations ago, but there were no accesses to it during the last $h-1$ iterations. In this case, the regions that can generate interferences with the attempt to reuse the SOL in the current iteration are those accesses during those h intermediate iterations, $\text{Reg}_{Ri}(h)$.

Example 14. When the reference `X(C(J))` in our example code of Figure 4.4 is analyzed in the context of the outer loop `I` at nesting level 0, the values read from the indirection are no longer guaranteed to be ordered throughout the execution of the loop. That is, the values read from `C` during a single iteration of the loop `I` correspond to the column indexes of the elements of a single row, which we can assume that have been stored in a given order; but when the whole loop `I` is taken into account, the values read from `C` are not ordered among different iterations of loop `I`. As a result, the non-monotonic irregular access PME of Equation (4.5) characterizes the access to `X` in this loop. In that equation, the number of iterations of the loop is $N_0 = M$ in our case, and $WM_{R0}(\text{RegIn}, j)$ is calculated following

Equation (4.6). In order to evaluate the latter equation we must calculate p_{R0}, that is, the individual probability each SOL of `X` is accessed in each iteration of loop `I`. As we have explained, this value is derived as $p_{R0} = L_{R0_1}/D_{R0}$, where L_{R0_1} is the number of different SOLs our reference accesses on average in each iteration of the loop, and D_{R0} is the number of different SOLs it could actually access. In example 12 we explained and calculated that for this reference $L_{R0_1} = L_{R1}$, and the value of L_{R1} was estimated in Equation (4.4) in example 13. Regarding D_{R0}, it is calculated according to Equation (4.3). As we explained in example 12, while the variable `I` that controls the loop we are analyzing does not appear in the expression of our reference `X(C(J))`, this variable indexes `R(I)`, which defines the values for `J`. This way, `I` indexes indirectly the indirection we are analyzing in the first (and only) dimension of array `X` and Equation (4.3) can be evaluated using the same parameters used in example 13, which results in $D_{R0} = D_{R1} = \lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$. That is, any of the $\lceil D_{\mathrm{X1}}/L_{\mathrm{s}} \rceil$ lines of `X` can be accessed during the execution of loop `I`, where we remind the reader that D_{X1} is the length of vector `X` and L_{s} is the number of elements of vector `X` a cache line can hold

This example helps us also illustrate the meaning and usage of the RegIn input for the PMEs. The PME F_{R0} for reference R =`X(C(J))` we have just built is based on Equation (4.5). In its development in Equation (4.6) we can see how, as always, this PME is expressed in terms of the PME for the same reference in the inmediately inner loop. In our case this PME is F_{R1}, built in example 13, which models the behavior of the accesses to `X` during the product by a row of the sparse matrix. The evaluations of $F_{R(i+1)}$ in F_{Ri} receive as RegIn the set of regions accessed during the reuse distance associated to that evaluation. In our example, attending to Equation (4.6), F_{R0} evaluates F_{R1} through $WM_{R0}(\mathrm{RegIn}, j)$ with two kinds of reuse distances. The input for the first appearance of F_{R1} in this expression depends on the RegIn for F_{R0} itself, because it is not associated to reuses within the loop. Rather, it corresponds to the first accesses to lines of `X` during the execution of the loop, which will result in cold misses. The model predicts this correctly because (a) RegIn for outer loops with no preceding accesses is a region with an associated miss probability 1 and (b) as we can see the model propagates this region down to the PME F_{R1} for the innermost loop for the evaluation of the misses generated in the very first accesses to these lines.

The remaining evaluations of F_{R1} in Equation (4.6) correspond to reuses within loop 0 with a reuse distance of exactly h iterations of this loop each. Such evaluations are multiplied by the probability this situation actually takes place to predict

correctly the number of misses they generate. Their RegIn is $\mathrm{Reg}_{R0}(h)$, i.e., the interference region generated during those h iterations in which a line of `X` has not been accessed. In our example this corresponds to the acesses that take place during the product of h rows of our sparse matrix by the vector. The RegIn of F_{R1} determines the miss probability for the first accesses to the lines of `X` during an isolated iteration of the innermost loop. Assigning this value to RegIn ensures such probability depends in fact on the cache footprint of the accesses performed since the inmediately preceding access to those lines, which took place exactly h iterations of the loop on `I` ago.

The calculation of regions of interference and the quatitative evaluation of PMEs are considered in Section 4.2.1. ■

4.2.3. Model Extension for Uniform Banded Matrices

Until now we have considered the case in which all the elements of the base array have the same probability of being accessed, but our model can be extended to cover situations in which the distribution is not uniform. For example, a very common source of indirections are accesses generated by sparse matrices that are stored in some compressed format like CRS [19]. One of the most usual situations by far is that such matrices are banded[1], so it is valuable to extend our model to consider irregular accesses that are are uniformly distributed in a limited band or area of the base array. In Section 4.3 we will describe a different model extension for banded matrices where the values are not uniformly distributed inside the band. In this case, the formulas described in the Sections 2.4 and 4.2.2 can be used making two small changes to adapt them to this new situation:

- when PMEs for the indirect accesses generated by the column indices of a banded matrix are built, the term $D_{\mathrm{A}j}$ in Equation (4.3) must be replaced by the size of the band of the studied matrix, since the accesses are not uniformly distributed on the whole j-th dimension of the base array, but only of the region associated with the band B of the matrix.
- since the nonzeros are only distributed along B rows in each column and B columns in each row, when the probability of reuse of a group of SOLs with respect to the preceding iterations is considered in Equation (4.5), the upper

[1] `A` is banded with bandwidth $B = 2p + 1$ if all the nonzeros are contained within the first p super and first p subdiagonals. $(A_{ij} = 0, |i - j| > p)$

```
DO I= 1,M
  DO K= R(I), R(I+1) - 1
    REG0=A(K)
    REG1=C(K)
    DO J= 1,H
      D(I,J)=D(I,J)+REG0*B(REG1,J)
    ENDDO
  ENDDO
ENDDO
```

Figure 4.5: Sparse Matrix - Dense Matrix Product with IKJ order

bound of the summatory is not N_i, the size of the sparse matrix along the considered dimension that gives place to the attempts of reuse, but B, since only along B rows/columns can be the same SOL of the base array be reused.

Example 15. The model derived for matrices with an uniform distribution for our example code in Figure 4.4 is applicable to banded matrices except in two points. First, in the calculation of D_{R0} and D_{R1} for reference `X(C(J))` we must substitute the value of D_{X1} with the band size. Also, in the expression F_{R0} that characterizes the behavior of this reference in the outer loop at nesting level 0 (the one indexed by `I`), which has the shape of eq. (4.5), the upper bound of the summatory must no longer be `M`, the total number of rows of the sparse matrix, but its band size B, since only along the processing of B different rows of the input matrix can we exploit reuse of a given line of the base array `X` of this reference. The size of the band would have to be provided by a directive to the compiler or be extracted by an analysis of the input data. ■

4.2.4. Validation

Our validation relies on eleven kernels of different complexities that contain indirections derived from the manipulation of sparse matrices stored in compressed formats such as the CRS [19] format. The first code is the Sparse Matrix - Vector Product (SPMXV) shown in Figure 4.4. The next three codes are the Sparse Matrix-Dense Matrix Product (SPMXDM) with the three different loop orderings this operation allows: IJK, JIK and IKJ, where the first index is the one for the outer loop and the last index the one for the innermost loop in the nest. In the three

```
1  DO I=2,N+1
     RT(I)=0
   END DO

2  DO I=1, R(M+1)-1
     J=C(I)+2
     RT(J)=RT(J)+1
   END DO

   RT(1)=1
   RT(2)=1

3  DO I=3, N+1
     RT(I)=RT(I)+RT(I-1)
   END DO
            ↪
```

```
4  DO I=1, M
     DO K=R(I), R(I+1)-1
       J=C(K)
       P=RT(J)
       CT(P)=I
       AT(P)=A(K)
       RT(J)=P+1
     END DO
   END DO
```

Figure 4.6: Transposition of a sparse matrix.

orderings I indexes the rows of the sparse matrix, K its columns, and J the columns of the dense matrix. As an example, the IKJ loop ordering is shown in Figure 4.5. Finally, Figure 4.6 shows a sparse matrix transposition (TRANSPOSE) where both the original and the transposed matrix are stored using the CRS method. This code is particularly complex, as it contains four loop nests, there are accessed with several levels of indirection in loop 4, and it involves more data structures than the other examples (six). Besides, some structures appear in several loop nests, so there may be reuses between the access to a line in one loop nest and another access in another loop nest.

The remaining five kernels have been extracted from the set of routines `matvec.f` of the well-known SPARSKIT [43] library. This set of routines contains different routines that perform the product between an sparse matrix and a vector, where the sparse matrix to multiply has been stored using different compressed storage formats. The routines analyzed are: AMUXMS, ATMUX, ATMUXR, AMUXD, AMUXE and AMUXJ. In AMUXMS the sparse matrix is stored in the MSR (Modified Sparse Row Storage) method; ATMUX and ATMUXR use again the CRS format but the input matrix is transposed; AMUXD uses a matrix stored in the DIA (Diagonal Storage Format) format; in AMUXE the matrix is stored in the ELL (Ellpack Itpack) format, and finally in AMUXJ it is stored using the JAD (Jagged-Diagonal Storage) format. All these storage formats are described in [43].

Code	$\overline{MR_{\text{Sim}}}$	$\overline{MR_{\text{Mod}}}$	$\overline{\Delta_{MR}}$	$\max(\Delta_{MR})$
SPMXV	9.64%	9.45%	0.92%	8.23%
SPMXDMIKJ	48.95%	47.92%	1.41%	11.48%
SPMXDMIJK	22.20%	21.42%	0.79%	3.56%
SPMXDMJIK	11.68%	11.28%	0.70%	6.65%
TRANSPOSE	18.98%	19.22%	1.60%	11.72%
AMUXMS	6.20%	5.91%	0.77%	8.78%
ATMUX	5.27%	4.82%	0.63%	11.20%
ATMUXR	5.24%	4.77%	0.61%	10.10%
AMUXD	4.62%	4.81%	0.78%	7.76%
AMUXE	5.69%	5.84%	0.37%	7.05%
AMUXJ	5.84%	6.47%	1.10%	9.97%

Table 4.1: Average measured ($\overline{MR_{\text{Sim}}}$) and predicted ($\overline{MR_{\text{Mod}}}$) miss rates, average value $\overline{\Delta_{MR}}$ of the absolute difference between the predicted and the measured miss rate in each experiment, and maximum value of this difference $\max(\Delta_{MR})$.

Validation with Synthetic Matrices

The integration of our model in the XARK compiler [15], which will be discussed in Chapter 5, has allowed us to apply it automatically to the validation kernels. The miss rate predicted by the model was compared with the results of trace-driven simulations using synthetic matrices with an uniform distribution of their nonzero elements. Over 10000 tests were performed for each code changing the sizes and starting addresses of the different arrays, the cache configuration and the density of the sparse matrix. Table 4.1 gives an idea of the accuracy of the model. Columns $\overline{MR_{\text{Sim}}}$ and $\overline{MR_{\text{Mod}}}$ contain the average values of the miss rate simulated and the miss rate predicted in the set of experiments, respectively. Then, column $\overline{\Delta_{MR}}$ contains the average value of the absolute value Δ_{MR} of the difference between the predicted and the measured miss rates for each experiment. We use absolute values, so that negative errors are not compensated with positive errors. Column $\max(\Delta_{MR})$ contains the largest value of Δ_{MR} observed in the set of experiments.

Tables 4.2, 4.3 and 4.4 show some random representative validation results for the Sparse Matrix - Vector Product, the Sparse Matrix - Dense Matrix Product with IKJ loop ordering and the Sparse Matrix Transposition codes, respectively, displaying a wide range of possible validation parameters and the result obtained.

In the three tables, the first two columns, M and N, show the number of rows

M	N	α	C_s	L_s	K	MR_{Sim}	MR_{Mod}	Δ_{MR}	T_{mod}
1000	1000	4.00	8K	32	1	30.11	30.00	0.11	0.015
1500	1100	12.12	32K	32	2	18.17	18.34	0.17	0.021
1600	1500	8.33	32K	64	4	8.44	8.60	0.15	0.010
1300	1400	13.74	64K	128	1	5.21	5.31	0.10	0.012
1700	1500	9.80	64K	64	2	8.67	8.82	0.15	0.032
1100	1000	22.73	128K	128	2	4.21	4.42	0.21	0.021
750	750	7.00	512K	128	8	4.23	5.13	0.90	0.014
5500	5500	0.28	1024K	64	8	8.77	8.86	0.09	0.035
3000	3000	1.19	2048K	128	4	4.26	5.64	1.38	0.033
1000	1200	16.67	128K	128	1	10.82	4.87	5.96	0.025

Table 4.2: Validation data and times for the Sparse Matrix - Vector Product code for several cache configurations, matrix sizes and sparse matrix density

M	N	α	H	C_s	L_s	K	MR_{Sim}	MR_{Mod}	Δ_{MR}	T_{mod}
900	900	22.22	500	32K	64	1	89.27	88.27	1.00	0.019
500	500	3.20	600	64K	64	4	81.97	81.61	0.36	0.011
700	700	31.43	500	64K	64	8	29.66	23.30	6.36	0.015
1100	1100	14.55	500	128K	64	8	30.76	29.88	0.88	0.027
1000	1000	15.00	750	128K	64	4	31.18	29.59	1.58	0.038
700	700	27.14	500	256K	64	2	21.25	20.71	0.54	0.019
1000	1000	24.00	500	512K	64	2	23.18	22.45	0.73	0.023
700	700	2.86	500	1024K	32	2	32.89	32.56	0.33	0.027
1000	1000	1.58	1000	2048K	64	4	38.10	36.13	1.97	0.052
600	600	30.00	500	32K	32	8	76.68	65.20	11.48	0.032

Table 4.3: Validation data and times for the Sparse Matrix - Dense Matrix Product IKJ code for several cache configurations, matrix sizes and sparse matrix density

M	N	α	C_s	L_s	K	MR_{Sim}	MR_{Mod}	Δ_{MR}	T_{mod}
600	600	35.00	16K	32	2	32.49	32.91	0.43	0.029
700	700	34.29	32K	32	1	28.02	26.14	1.89	0.025
3000	2000	2.50	64K	32	2	27.00	29.86	2.86	0.031
5000	2000	3.00	64K	128	1	26.53	27.47	0.93	0.027
1000	1000	15.00	128K	128	1	17.77	19.21	1.44	0.035
800	800	57.50	256K	64	4	5.35	5.17	0.18	0.034
500	500	46.80	512K	128	8	2.18	3.00	0.82	0.037
2900	2900	0.47	1024K	64	1	7.91	10.29	2.38	0.043
500	500	15.75	2048K	64	4	3.08	4.36	1.28	0.042
5000	1000	9.00	128K	64	4	11.50	23.22	11.72	0.023

Table 4.4: Validation data and times for the Matrix Transposition code, for several cache configurations, matrix sizes and sparse matrix density

and columns of the sparse matrix involved in the code, respectively. Then, column α is the density or percentage of positions in the sparse matrix with nonzeros. In table 4.3, column H shows the number of columns of the dense matrix involved in the product. The cache configuration is given in the three tables by C_s, the cache size in bytes, L_s, the line size in bytes, and K, the degree of associativity of the cache. Larger cache lines and associativities tend to be associated to larger caches in general in the tables, as this is the most common situation. For each combination of the input problem parameters and cache configurations the tables display the miss rate MR_{Sim} measured by the simulations, the miss rate MR_{Mod} predicted by our model, and Δ_{MR}, the absolute value of the difference between them. These three values are expressed as percentages between 0 and 100. The last entry in every table contains the data for the experiment that generated the largest Δ_{MR}.

Finally, the last column in the three tables, T_{mod}, reflects the corresponding modeling times in seconds in a 2,08 GHz AMD K7 processor-based system, respectively. Modeling times, which were always below one second, are several orders of magnitude shorter than trace-driven simulation for the sparse matrix-dense matrix products, and noticeably shorter in the case of the other codes.

Validation with real banded matrices

In order to validate our model for uniform banded matrices we used the Sparse Matrix - Vector Product code shown in Figure 4.4, the Sparse Matrix - Dense Matrix

Matrix Name	Size	B	α	C_s	L_s	K	MR_{Sim}	MR_{Pred}	Δ_{MR}	T_{mod}
jpwh991	991	155	0.61	64K	64	4	9.37	8.84	0.53	0.014
jpwh991	991	155	0.61	32K	32	2	18.77	17.72	1.05	0.013
jpwh991	991	155	0.61	32K	64	1	10.29	9.84	0.45	0.012
bcsstk05	153	20	10.35	32K	64	1	9.57	9.11	0.46	0.009
bcsstk05	153	20	10.35	256K	16	4	35.04	34.13	0.91	0.009
bcsstk05	153	20	10.35	256K	32	2	17.54	17.07	0.48	0.009
bcsstm10	1086	71	1.87	32K	64	1	9.12	9.29	0.17	0.013
bcsstm10	1086	71	1.87	256K	16	4	34.66	33.91	0.74	0.017
bcsstm10	1086	71	1.87	1024K	64	4	8.67	8.48	0.19	0.015
jpwh991	991	155	0.61	8K	16	1	43.79	40.45	3.33	0.013

Table 4.5: Validation data and times for the Sparse Matrix - Vector Product code for several cache configurations and different Harwell-Boeing matrices with uniform band distribution

Matrix Name	Size	B	α	H	C_s	L_s	K	MR_{Sim}	MR_{Pred}	Δ_{MR}	T_{mod}
jpwh991	991	155	0.61	200	32K	64	1	93.08	93.06	0.02	0.011
jpwh991	991	155	0.61	153	16K	32	2	88.61	88.27	0.33	0.010
jpwh991	991	155	0.61	1086	32K	32	4	97.30	98.52	1.21	0.017
jpwh991	991	155	0.61	350	64K	64	4	91.26	92.10	0.83	0.011
bcsstk05	153	20	10.35	153	32K	32	4	16.49	16.84	0.35	0.009
bcsstk05	153	20	10.35	153	16K	32	2	45.34	43.30	2.04	0.009
bcsstm10	1086	71	1.87	153	16K	64	4	74.22	74.32	0.10	0.010
bcsstm10	1086	71	1.87	153	32K	128	1	63.73	62.59	1.13	0.011
bcsstm10	1086	71	1.87	153	512K	64	4	0.70	0.65	0.05	0.014
bcsstm10	1086	71	1.87	200	1024K	64	8	0.68	0.55	0.13	0.058
bcsstk05	153	20	10.35	350	32K	64	1	72.96	61.30	11.67	0.011

Table 4.6: Validation data and times for the Sparse Matrix - Dense Matrix Product IKJ code for several cache configurations and different Harwell-Boeing matrices with uniform band distribution

Product in Figure 4.5, and the Sparse Matrix Transposition in Figure 4.6 and we applied them to real matrices from the Harwell-Boeing collection [28] rather than to synthetic matrices. The results of some randomly chosen validation experiments are shown in Tables 4.5 and 4.6 for the first two codes considered, respectively. In both tables the first columns contain the name of the matrix used in every test, followed by the characteristics of the matrix such as, the number of rows and columns size (we used square matrices), the band size B and, in the case of sparse matrix - dense matrix product code, the number of columns H of the dense matrix. α is the percentage of positions in the sparse matrix with nonzeros. The used cache configuration C_s, L_s and K, follows. Again, for each experiment we show both the measured MR_{Sim} and the predicted MR_{Mod} miss rates and the absolute value of the difference between them, Δ_{MR}. Many different experiments were performed using different cache configurations; the results shown in these tables are only a small representative subset of these tests. The last entry in every table contains again the data for the experiment that generated the largest Δ_{MR}.

For the sparse matrix - vector product code we performed 510 different tests changing the used matrix, the cache configuration, and the base address of the data structures involved in the code, obtaining an average value for the Δ_{MR} of 0.66% and a maximum value of 3.33%, the average value of the relative error Δ^R_{MR} was 3.96%. The metric Δ^R_{MR} stands for the relative error of our prediction: it is the absolute value of the difference between the miss rate measured by the simulation and the miss rate predicted by the model (Δ_{MR}) divided by the miss rate measured by the simulation and expressed as a percentage, that is $\Delta^R_{MR} = \Delta_{MR}/MR_{\text{Sim}} \times 100$.

For the sparse matrix - dense matrix product code we performed 5100 different tests, changing the same parameters as for the sparse matrix - vector product code as well as the number H of columns of the dense matrix involved in the code. We obtained an average value for Δ_{MR} of 2.55% and a maximum value of 11.67%. The average value of the relative error Δ^R_{MR} was 6.60%.

Finally, we performed the same set of 510 tests for sparse matrix transposition as for sparse matrix-vector product. In this case the average Δ_{MR} was 1.78%, and its maximum was 7.30%, being the average value of the relative error Δ^R_{MR} 11.35%.

Again, these validation results obtained using a wide range of parameter combinations, and which are very similar to the ones obtained for the model with a completely uniform distribution displayed in Table 4.1, make us think that our model is a good estimator of the behavior of a code with irregular access patterns

under the assumed conditions.

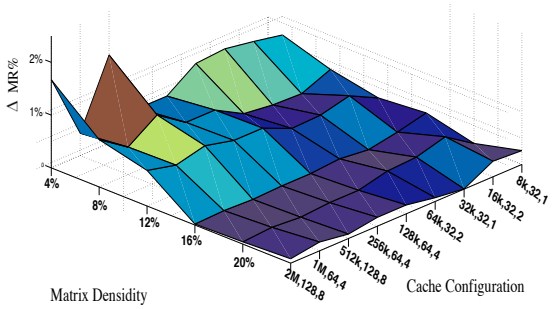
Finally, as in the previous tests, the last column in Tables 4.5 and 4.6, T_{mod}, represents the time consumed by our model. The model is several orders of magnitude faster than the simulation.

Discussion

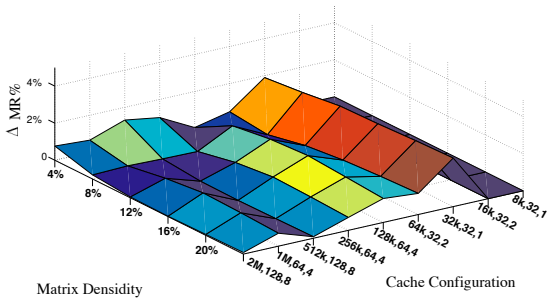
The model worked well for the sparse matrix - vector product of Figure 4.4. The results were somewhat worse for the sparse matrix - dense matrix product code in Figure 4.5 for both kinds of matrices, although the model was still very accurate in general. Predicting the reuse for the reference `B(C(K),J)` that generates irregular accesses in this code is possibly more complex than for the references subject to irregular access patterns in the other codes. The reason is that in this case each value of the indirection controls a whole set of tightly coupled accesses of `B(C(K),J)` to different lines with a regular stride for `J`=1, ..., `H`, while in the other codes each individual indirection only controls the access to one line. It is good to see that in such a complex situation, the predictions of the model are still good. The behavior of the model for the sparse matrix-dense matrix products in which the inner loop is `K` is similar to the one observed for the sparse matrix - vector product as we see in Table 4.1. Finally, the transposition of a sparse matrix in Figure 4.6 turned out to be the most difficult code to predict, as it is not a perfectly nested loop like the previous examples, and it displays several levels of indirection in its fourth loop. Still, the predictions of the model were very reasonable.

The tendencies of the accuracy of the model with respect to the parameters of the caches and the density of the sparse matrix are displayed in Figure 4.7, in which we have used cache configurations that are similar or equal to real level 1 and level 2 caches of current computers. Cache configurations are expressed as C_{s},L_{s},K, where C_{s} is the cache size in bytes, L_{s} is the line size in bytes and K is the associativity. The most important conclusion is that in general, higher densities lead to more accurate predictions. That is an expected result, since the lower density, the more irregular the accesses. Also, notice that this higher irregularity leads to higher miss rates (as an example, see experiment in Figure 4.8), which dilute the larger values of Δ_{MR}.

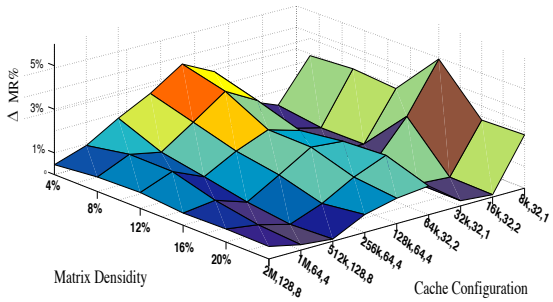
As for the time required to compute its predictions, the model takes more time when the size of the problem (size of the involved data structures) is bigger, as ex-



(a) Sparse matrix-vector product, with $M = N = 750$



(b) Sparse matrix-dense matrix product (IKJ), with $M = N = H = 750$



(c) Sparse matrix transposition, with $M = N = 750$

Figure 4.7: Δ_{MR} as a function of the sparse matrix density and the cache configuration in different codes. Cache configurations are expressed as C_s,L_s,K, where C_s is the cache size in bytes, L_s is the line size in bytes and K is the associativity

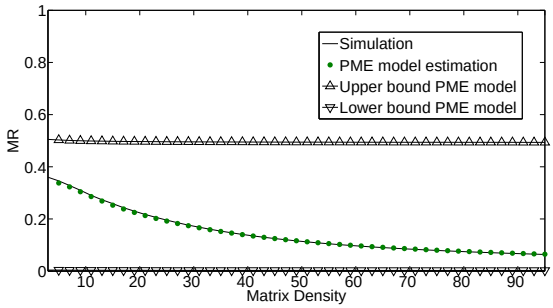


Figure 4.8: Miss rate measured and predicted following different strategies as a function of the matrix density for the sparse matrix-dense matrix product (IKJ), where $M = N = H = 500$ in a cache of 64Kbytes with a line size of 64 bytes and associativity degree 4.

pected, and when the cache associativity is higher. The reason for the latter behavior is that the complexity algorithm for calculating the area vector for some patterns depends directly on this argument. Still, modeling times are always below one second. In general, we can say that our model provides quite accurate estimations with a very low computing cost.

The sparse matrix-dense matrix product with IJK loop ordering is used in Figure 4.8 to compare the miss rate obtained by a trace-driven simulation, the miss rate predicted by the PME model, an upper bound of the prediction obtained by a simplified version of our model that considers all the irregular accesses as misses, and a lower bound obtained by ignoring the irregular accesses that appear in the code. The sizes of the data structures involved in the code and the cache configuration were kept constant while the density of the sparse matrix took values between 1% and 100%. The figure reflects that the PME model estimates the miss rate accurately, while simplified versions provide very poor estimations. This justifies the interest of our model.

A more detailed study of how changes in any of the cache configuration parameters can affect the cache performance can be performed for any code. For this purpose we considered the AMUXMS code which performs the product between an sparse matrix and a vector, the base cache configuration has a total size of 512 KBytes a line size of 32 bytes and an degree of associativity of 4. We tracked the evolution of the miss rate measured and the miss rate predicted by the model changing separetely each one of these parameters. The results of these experiments, reflected in Figure 4.9, were obtained using an square sparse matrix of 500x500 and 50000

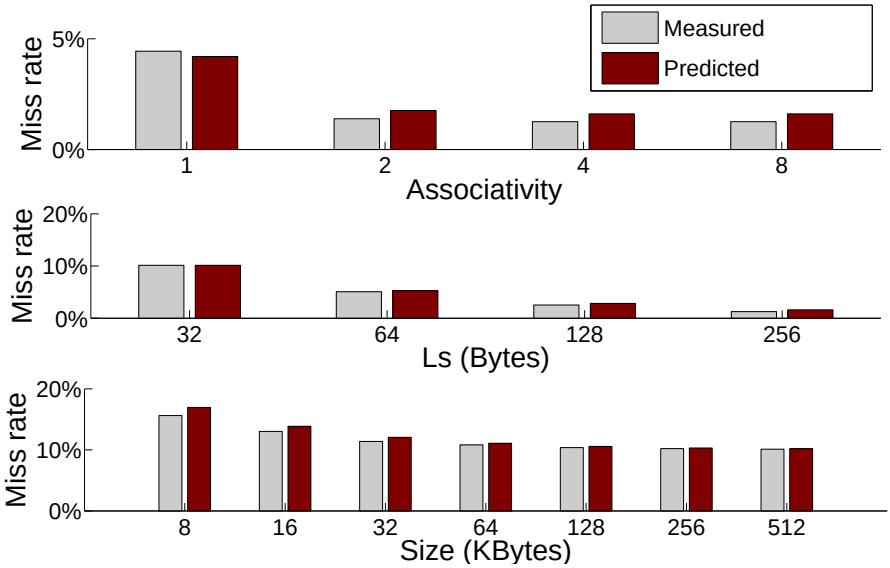


Figure 4.9: Miss rate measured and miss rate predicted for the AMUXMS code. In the first graphic the associtivity degree is changed; the second graphic modifies line size; the third graphic considers different caches sizes.

non-zero values uniformly distributed along the matrix. In the first experiment the degree of associativity takes values 1, 2, 4 and 8 respectively. The direct-mapped cache has a bigger miss rate than those caches with larger degrees of associativity. This improvement decreases as the associativity grows. Even, in some cases for large associativities like 8, there is a slight performance reduction. The second experiment considers line sizes of 32, 64, 128 and 256 bytes respectively. A bigger line size produces a significant miss rate decrease. But when the line size is big enough this improvement is attenuated because although bigger lines reduce the number of cold misses, very big lines can increase the interference between different data structures stored in the cache. The third experiment changes the total cache size which takes values of 8, 16, 32, 64, 128, 256 and 512 KBytes respectively. Always, the bigger the cache size, the smaller the miss rate because there is more room for storing the data structures managed by the program, but this effect is diminished in very big caches. The reason is that there is less and less room for improvement as the cache size approaches the problem size. In all these experiments the model predictions were very accurate.

We also, compared the cache behavior when the different analyzed codes to perform an sparse matrix-vector product were ran. We consider an sparse matrix of 1000x1000 with 100000 non-zero values uniformly spread along the matrix. The com-

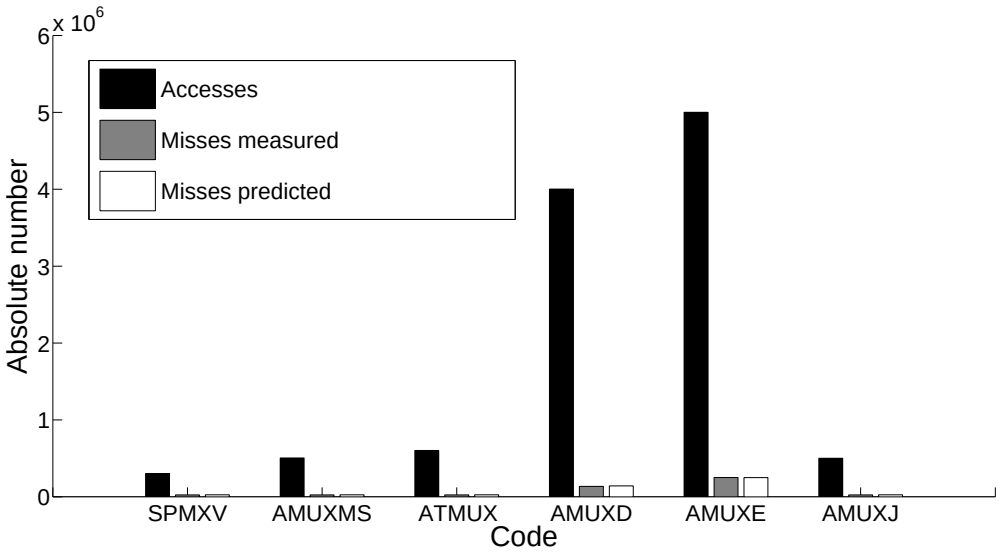


Figure 4.10: Number of accesses, number of misses measured and predicted for an sparse matrix-vector product using different compressed storage formats. The cache configuration considers a cache size of 32 KBytes, a line size of 64 bytes and an associativity degree of 4.

pared codes are the SPMXV, AMUXMS, ATMUX, AMUXD,AMUXE and AMUXJ (ATMUXR is omitted because its characteristics are very similar to ATMUX as it was seen in Table 4.1). These codes perform an sparse matrix-vector product when the matrix is stored using different compressed storage methods. The graph in Figure 4.10 represents the number of accesses, the number of misses measured and the number of misses predicted by the model for each code. The prediction of the model are always very accurate. It can be seen how some codes like the AMUXD and AMUXE perform much worse than the other ones because they are designed for banded matrices with very few diagonals, so their performance (in terms of cache misses) when applied to an uniform matrix is really poor. The other codes obtain a very similar performance.

Finaly, we have also inquired into what happens when the model is applied to matrices with a non-uniform distribution of the entries. In order to quantify this behavior, we run experiments on 320 randomly chosen matrices from the Harwell-Boeing [28] and NEP [18] collections, using 10 different cache configurations for each one with sizes ranging from 16 KBytes to 2 MBytes, thus yielding a total of 3200 experiments per analyzed kernel. Figure 4.11 summarizes the results of these experiments classifying our experiments in four buckets according to the Δ_{MR}

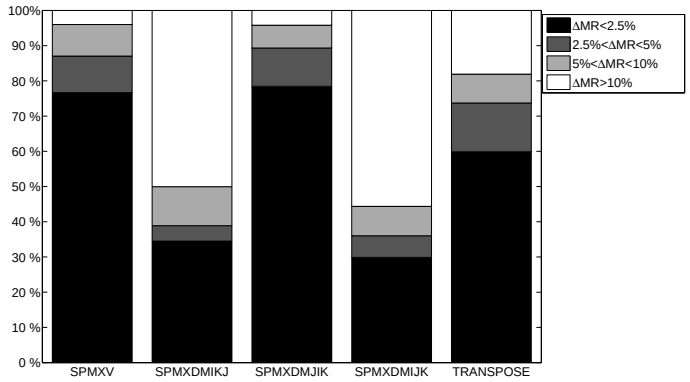


Figure 4.11: Percentage of the number of experiments in which the Δ_{MR} is below 2.5%, between 2.5% and 5%, between 5% and 10%, or larger than 10% when real matrices with a non-uniform distribution of the entries are used.

achieved: below 2.5%, between 2.5% and 5%, between 5% and 10%, and larger than 10%. We see that SPMXV, SPMXDM with JIK loop ordering and TRANSPOSE yield reasonable estimations in the vast majority of the cases, while SPMXDM with the IKJ and IJK orderings is less reliable. When irregular accesses are not uniformly distributed, they tend to be grouped in clusters, which increases the locality. So in these cases, our model can still help understand the behavior of the cache, although the miss rate it predicts must be considered an upper bound rather than an accurate estimation.

4.3. Model Extension for Non-Uniform Banded Matrices

Most real data involved in irregular computations due to the existence of indirections does not follow an uniform distribution. The banded distribution is an example of non-uniform distribution present in many matrices. This distribution is very common in sparse computations, the main source of the codes used in the validation of our model. As we saw in the validation in Section 4.2.4 the model PME extension for codes involving uniform banded matrices is not suitable for the modeling of matrices with a non-uniform distribution of the values inside the band.

The modeling of this kind of non-uniform distributions is very complex. The equations for references with regular access patterns are relatively simple because all the accesses that can result in a cold miss have an unique interference probability,

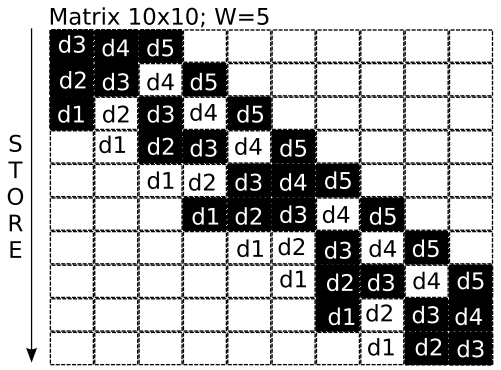


Figure 4.12: Banded sparse matrix

and a different unique interference probability is applied for the accesses that can result in an interference miss, as all the reuses have the same constant reuse distance.

In an irregular pattern, every access has a set of different possible reuse distances with an associated interference probability that is weighted with the probability that each considered reuse attempt happens. If the distribution of the accesses is uniform, the same set of interference regions can be used for all the accessed lines and they all have the same probability of reuse associated to each reuse distance. When this is not the case, that is, when different lines have different probabilities of being accessed, a different set of interference regions must be calculated for each accessed line and different lines will have different probabilities of reuse for the same reuse distance.

We will illustrate these ideas with the code in Figure 4.4, which performs the product between a sparse matrix stored in CRS format [19] and a vector, and which is part of SPARSKIT [43]. Let us remember that the CRS format stores sparse matrices by rows in a compressed way using three vectors. One vector stores the nonzeros of the sparse matrix ordered by rows, another vector stores the column indexes of the corresponding nonzeros, and finally another vector stores the position in the other two vectors in which the data of the nonzeros of each row begins. In our codes we always call these vector `A`, `C` and `R` respectively. The innermost loop of the code in Figure 4.4 performs the product between vector `X` and row `I` of the sparse matrix. In this code reference `X(C(J))` performs an irregular access on vector `X` only in the positions in which the matrix row contains nonzeros. Let us suppose that the sparse matrix that is being multiplied is a banded matrix like the one shown in Figure 4.12, in which the $W = 5$ diagonals that constitute its band

have been labeled and black and white elements represent non-zero and zero values elements, respectively. During the processing of each row of the sparse matrix, a maximum of W different elements of `X` will be accessed. Each one of these W elements has a different probability of being accessed that depends on the density of the corresponding diagonal in the banded matrix. The set of elements eligible for access is displaced one position in the processing of each new row. Also, each element of `X` will be accessed a maximum of W times during the execution of the code, as a maximum of W rows may have nonzeros in the corresponding column. Interestingly, the probability of access is not uniform along those W rows. For example, every first potential access during the processing of this matrix in this code will take place for sure, while every second potential access to an element of `X` will happen with a probability of 30%. This is because all the positions in the fifth diagonal (d_5) keep nonzeros, while in the fourth diagonal (d_4) of the band 3 out of its 9 positions keep nonzeros, which is a density of nonzeros of 30%

The number of elements of the vector accessed in the processing of a row can be averaged using the densities of the diagonals of the band matrix. Every L_s elements of the vector are stored in a different line, the probability of accessing that line can be calculated as a function of the corresponding densities in the diagonals of the sparse matrix. In every iteration of the outermost loop, a different row of the sparse matrix is selected. It is possible to reuse lines of the vector between different iterations of the outermost loop. In the processing of a given row of the sparse matrix some values of the vector mapped inside the band are accessed, in the processing of the next row the situation is repeated but the area covered by the band is shifted one position to the right, this has to be taken in account for the calculation of the possible reuse between the processing of different rows.

The situation depicted in our example is clearly more common than the extension performed in Section 4.2.3, in which we only considered irregular access patterns which had an uniform probability of access for each element of the dereferenced data structure, and in which such probability did not change during the execution of the code. It is very usual that the diagonals of banded matrices have different densities, with the distribution of the nonzeros within each diagonal being relatively uniform. As a result, we have extended our model to cope with this important class of matrices, which enables to model automatically and accurately the cache behavior of codes with irregular access patterns in the presence of a large number of real sparse matrices, as the evaluation proves. We will characterize the distribution of nonzeros in these matrices by a vector $\vec{d}$ of W probabilities where d_i contains the

density of the $i-th$ diagonal, that is, the probability a position belonging to the $i-th$ diagonal of the band contains a nonzero. This extension can be automated using a compiler framework that satisfies its information requirements. The vector $\vec{d}$ of diagonal densities is the only additional information we need in this work. These values are obtained from an analysis of the input data that can be provided by the user, or obtained by means of runtime profiling.

Section 4.3.1 contains the new equations added to cover this situation, while Section 4.3.2 contains a description of the validation of this extension.

4.3.1. PME equations for Banded Matrices

The PMEs are a function of input memory regions calculated in outer or preceding loops that are associated to the reuses of the sets of lines (SOLs) accessed by R in loop i whose immediately preceding access took place before the loop began its execution. The uniformity of the accesses in all our previous extensions for covering irregular computation allowed to use a single region Reg for this purpose, that is, all the SOLs had the same reuse distance whenever a loop began. This happened because all the considered lines had uniform probabilities of access, and thus they also enjoyed equal average reuse distances and miss probabilities. The lack of uniformity of the accesses makes it necessary to consider a separate region of interference for each SOL. Thus we extend the PMEs to receive as input a vector $\vec{Reg}$ of memory regions. The element Reg_l of this vector is the memory region accessed during the reuse distance for what in this level of the nest happen to be first access to the l-th SOL that R can access. Another way to express it is that Reg_l is the set of memory regions that could generate interferences with an attempt to reuse the l-th SOL right when the loop begins its execution. This way, $\vec{Reg}$ has as many elements as SOLs defines R during the execution of the considered loop.

The shape of PME F_{Ri} depends on the access pattern followed by R in loop i. This section contains a description of the formulas we have developed for references with irregular access patterns generated by indirections due to the compressed storage of banded matrices in which the distribution of non-zeros within the band is not uniform. A different formula will be applied depending on whether the values read from the index array are known to be monotonic or not. They are monotonic when, given two iterations of the current loop i and j and being $f(i)$ and $f(j)$ the values generated by the index array in these iterations, for all $i \leq j$ then $f(i) \leq f(j)$ or for all $i \leq j$ then $f(i) \geq f(j)$. When the index values are known to be monotonic

a more accurate estimation can be obtained because we known that if our reference R reuses a SOL of the base array in a given iteration, this SOL is necessarily the one accessed in the previous iteration of the loop.

PME for irregular monotonic access with non-uniform band distribution

If we assume that the nonzeros within each row have been stored ordered by their column index in our sparse matrix in CRS format, reference `X(C(J))` generates a monotonic irregular access on the base array `X` during the execution of the innermost loop in Figure 4.4. Let us remember that the index array `C` stores the column indexes of the nonzeros of the row of the sparse matrix that is being multiplied by `X` in this loop.

The general equation that estimates the number of misses generated by a reference R in nesting level i that exhibits an irregular monotonic access with a non-uniform band distribution is

$$F_{Ri}(\vec{Reg}) = \left(\sum_{l=0}^{L_{Ri}-1} p_{i(lG_{Ri})} F_{R(i+1)}(Reg_l) \right) + \left(\sum_{l=1}^{W} d_l - \sum_{l=0}^{L_{Ri}-1} p_{i(lG_{Ri})} \right) F_{R(i+1)}(IntReg_{Ri}(1)) \tag{4.7}$$

The interference region from the outer level is different for each set of lines (SOL) accessed and it is represented as a vector $\vec{Reg}$ of L_{Ri} different components, where L_{Ri} is the total number of different SOLs of the base array `A` that R can access in this nesting level. L_{Ri} is calculated as $\lceil W/G_{Ri} \rceil$ being W the band size and G_{Ri} is the average number of positions in the band that give place to accesses of R to a same SOL of the base array `A`. This value is calculated as $\lceil L_s/S_{Ri} \rceil$, being $S_{Ri} = \alpha_{Rj} \cdot d_{Aj}$ where j is the dimension whose index depends on the loop variable $\mathtt{I}_i$ through the indirection; L_s is the cache line size; α_{Rj} is the scalar that multiplies the index array in the affine function, and d_{Aj} is the cumulative size[2] of the j-th dimension of the array `A` referenced by R.

Example 16. If we consider reference `X(C(J))` in Figure 4.4, while processing the matrix in Figure 4.12, with a cache line size $L_s = 2$, in the innermost level $d_{A1} = 1$

[2]Let `A` be an N-dimensional array of size $D_{A1} \times D_{A2} \times \ldots D_{AN}$, we define the cumulative size for its j-th dimension as $d_{Aj} = \prod_{i=1}^{j-1} D_{Ai}$

and $\alpha_{R1} = 1$. Each $G_{Ri} = 2$ consecutive positions in the band give place to accesses to the same SOL of `X`. Consequently, since $W = 5$, the number of different SOLs of `X` accessed would be $L_{Ri} = \lceil 5/2 \rceil = 3$. ■

The vector of probabilities $\vec{p_i}$ has W positions. Position s of this vector keeps the probability that at least one of the diagonals s to $s + G_{Ri} - 1$ has a nonzero, that is, it is the probability they generate at least one access to the SOL of the base array that would be accessed if there were nonzeros in these diagonals. Each component of this vector is computed as :

$$p_{is} = 1 - \prod_{l=s}^{min(W, s+G_{Ri}-1)} (1 - d_l) \tag{4.8}$$

Let us remember that $\vec{d}$ is a vector of W probabilities, d_s being the density of the $s - th$ diagonal in the band as it is reflected in Figure 4.12.

In Equation 4.7 each SOL l of the base array that `R` can access in nesting level i has a probability $p_{i(lG_{Ri})}$ of being accessed, where lG_{Ri} is the first band that can generate accesses to the $l - th$ SOL. The miss probability in the first access to each SOL l depends on the interference region from the outer level associated to that SOL Reg_l. The remaining accesses are non-first accesses during the execution of the loop, and because the access is monotonic, their reuse distance is necessarily on iteration of the loop. As a result, the interference region will be $IntReg_{Ri}(1)$, the interference region of reference R in 1 iteration of nesting level i. The number of potential reuses of SOLs by `R` in the loop is calculated as $\sum_{l=1}^{W} d_l - \sum_{l=0}^{L_{Ri}-1} p_{i(lG_{Ri})}$, where the first term estimates the number of different accesses generated by `R` during the processing of a row or a column of a band while the second term is the average number of different SOLs that `R` accesses during this processing.

Example 17. Examples 13 and 14 contained the derivation of the PME equations that describe the cache behavior of the reference `X(C(J))` in the code of Figure 4.4 in loops `I` and `J` respectively, assuming that the values generated by the index array `C` follow an uniform distribution. This code performs the product between a matrix, stored in CRS format, and a vector `X`. In example 13 it was established that the values generated by the index array during each complete execution of the innermost loop are monotonically increasing. If we consider that the CRS matrix used as input data is a banded matrix, then the Equation(4.7) must be applied. As loop `J` is the

innermost level for this reference, the resulting equation is,

$$F_{R1}(\vec{Reg}) = \left(\sum_{l=0}^{L_{R1}-1} p_{1(lG_{R1})} AV_0(Reg_l) \right) + \left(\sum_{l=1}^{W} d_l - \sum_{l=0}^{L_{R1}-1} p_{1(lG_{R1})} \right) AV_0(IntReg_{R1}(1)) \tag{4.9}$$

where L_{R1} is calculated as $\lceil W/G_{R1} \rceil$, being $G_{R1} = \lceil L_s/S_{R1} \rceil$. The stride is $S_{R1} = d_{X1} = 1$ as the array `X` is indexed using the affine function 1×`C`(I)+0. The probabilities p_{1l} are calculated using the Equation(4.8) ■

PME for irregular non-monotonic access with non-uniform band distribution

A data structure stored in a compressed format, such as CRS [19], is typically accessed using an offset and length construction [39]. In this situation, very common in sparse matrix computations, the knowledge that the values accessed across the indirection follow a banded distribution can be used to increase the accuracy of the prediction using a specific equation. For example, in the code of Figure 4.4 the reference `X`(`C`(`J`)) accesses the structure `C` using an offset and length construction. The values generated by the index array `C` in the innermost loop are monotonic but the values read across different iterations of the outermost loop are non-monotonic because a different row is processed in each iteration of this loop. When this situation is detected and we are in the presence of a banded matrix, the behavior of the reference in the outer loop can be estimated as

$$F_{Ri}(RegIn) = N_i F_{R(i+1)}(\vec{Reg}(RegIn)) \tag{4.10}$$

In this equation the N_i iterations in the current nesting level are considered to repeat the same behavior. Although the $W-1$ first and last iterations have a different behavior than the others as for example their band is not W positions wide, we have checked experimentally that the lost of accuracy incurred when not considering this is not significant. This is expected, as usually the band size W is much smaller than N_i, which is the number of rows or columns of the sparse matrix.

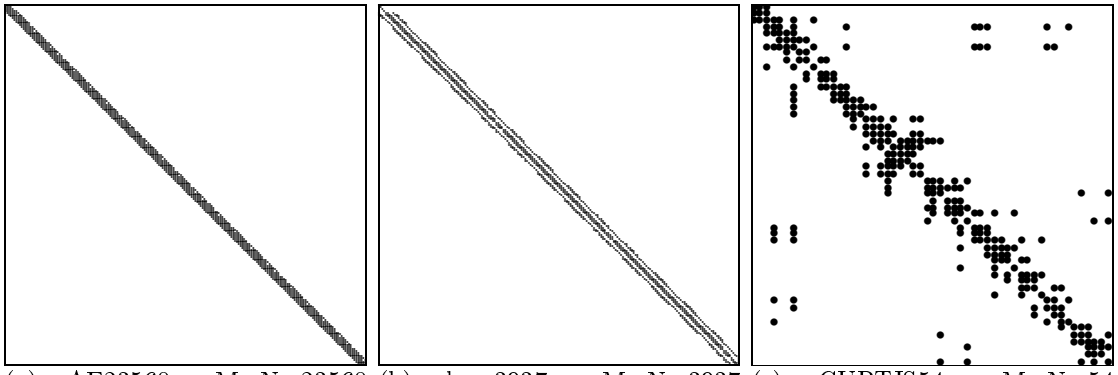
An average interference region for each one of the L_{Ri} SOLs accessed in the inner level must be calculated. This average interference region takes account of all the

possible reuses that can take place with respect to a previous iteration of the current loop depending on the different possible combinations of accesses to the studied base array. The interference region associated with each possible reuse distance must be added to the average region weighted with the probability an attempt of reuse with this reuse distance happens. The expression that estimates the interference region associated to the $l-th$ SOL that R can access in this loop is,

$$\begin{aligned} Reg_l(RegIn) = \prod_{z=lG_{Ri}+1}^{W}(1-p_{iz})(RegIn \cup IntReg_{Ri}(W-lG_{Ri}-1)+ \\ \sum_{s=lG_{Ri}+1}^{W} p_{is}\left(\prod_{z=lG_{Ri}+1}^{s-1}(1-p_{iz})\right) IntReg_{Ri}(s-lG_{Ri}) \end{aligned} \tag{4.11}$$

In the previous section we saw that lG_{Ri} is the first diagonal that could generate an access to the l-th SOL in a given iteration and $p_{i(lG_{Ri})}$ the probability of accessing that SOL during the processing of a row or column of the matrix. As the band is shifted one position to the right every row, in general, the probability that the same SOL of the base array is accessed by R m iterations before the current iteration is $p_{i(lG_{Ri}+m)}$. As a result, $\prod_{z=lG_{Ri}+1}^{W}(1-p_{iz})$ calculates the probability that the $l-th$ SOL has not been accessed in any previous iteration of this loop. In this case the interference region is equal to the union of the input region from the outer level and the region associated to the accesses that take place in the $W-lG_{Ri}-1$ previous iterations. The addition of a region to the average region weighted by its corresponding probability is performed adding the region weighted by the corresponding probability to the average region. Regarding the reuses within loop i, the probability that the last access to a SOL took place exactly m iterations ago is calculated multiplying the probability of being accessed in that iteration $p_{i(lG_{Ri}+m)}$ by the product of the probabilities of not being accessed in any of the iterations between that iteration and the current iteration $\prod_{z=lG_{Ri}+1}^{lG_{Ri}+m-1}(1-p_{iz})$. The interference region associated to this attempt of reuse will be the region covered by the accesses that take place in those m iterations of the current loop. In this equation $L_{Ri} = L_{Rj}$, $G_{Ri} = G_{Rj}$ and the vector $\vec{p_i} = \vec{p_j}$, being j the innermost nesting level of the offset and length construction.

Example 18. In example 14 the access done in the code in Figure 4.4 by the reference `X(C(J))` in the outermost loop was determined to be non-monotonical. When



(a) AF23560 M=N=23560 nnz=484256 W=609 (b) lnsp3937 M=N=3937 nnz=25407 W=168 (c) CURTIS54 M=N=54 nnz=291 W=43

Figure 4.13: Examples of matrices in the Harwell-Boeing set, M and N stands for the matrix dimension, nnz is the number of nonzeros and W is the band size.

the CRS matrix is banded the Fomula 4.3.1 must be applied, as a result,

$$F_{R0}(RegIn) = MF_{R1}(\vec{Reg}(RegIn)) \tag{4.12}$$

As the innermost nesting level of the offset and length is level 1, $L_{R0} = L_{R1}$,$G_{R0} = G_{R1}$ and $\vec{p_0} = \vec{p_1}$, so they take the same values calculated in Example 17. The L_{R0} values of the vector $\vec{Reg}(RegIn)$ are calculated using Equation 4.11.

4.3.2. Validation for Codes with Non-Uniform Banded Matrices

The validation was done applying by hand the PME model to: the sparse-matrix vector product, in Figure 4.4, the sparse-matrix dense-matrix product with IKJ (see Figure 4.5), IJK and JIK order, and the sparse-matrix transposition, shown in Figure 4.6.

The model was validated again comparing its predictions with the results of trace-driven simulations. For every code, 10 different cache configurations were tried with caches of sizes from 16 KBytes to 2 MBytes, line sizes from 16 to 64 bytes and associativity degrees 1, 2, 4 and 8. The input data set were the 177 matrices from the Harwell-Boeing [28] and the NEP [18] sets that we found to be banded or mostly banded (a few nonzeros could be outside the band). These matrices represent 52% of the total number of matrices contained in these collections.

Code	$\overline{MR_{\text{Sim}}}$	$\overline{\sigma_{\text{Sim}}}$	Uniform Bands Model		Non-Uniform Bands Model	
			$\overline{MR_{\text{Mod}}}$	$\overline{\Delta_{MR}}$	$\overline{MR_{\text{Mod}}}$	$\overline{\Delta_{MR}}$
SPMXV	14.00%	0.08%	15.57%	1.80%	14.45%	0.70%
SPMXDMIKJ	27.66%	2.02%	45.62%	26.81%	28.85%	4.19%
SPMXDMIJK	8.62%	0.29%	27.48%	17.23%	10.91%	3.10%
SPMXDMJIK	7.87%	0.43%	10.63%	3.23%	8.36%	0.78%
TRANSPOSE	10.31%	0.33%	11.38%	3.55%	9.52%	3.23%

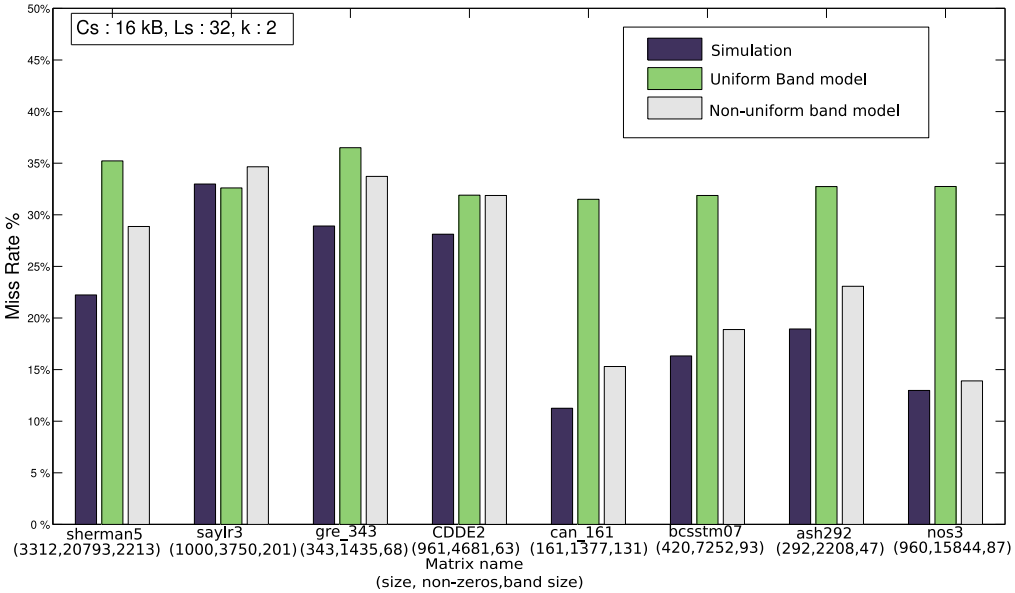
Table 4.7: Average measured ($\overline{MR_{\text{Sim}}}$) miss rate, average typical deviation ($\overline{\sigma_{\text{Sim}}}$) of the measured miss rate, average predicted ($\overline{MR_{\text{Mod}}}$) miss rate and the average value $\overline{\Delta_{MR}}$ of the absolute difference between the predicted and the measured miss rate in each experiment.

The matrices used are a heterogeneous set of input data. Some matrices have all their entries uniformly spread along a band, like the AF23560 matrix in Figure 4.13(a). The LNSP3937 matrix shown in Figure 4.13(b), has all its values spread along a band of the matrix but not uniformly. Finally, there are some matrices like CURTIS54, shown in Figure 4.13(c), where not all the values are spread along a band but a significant percentage of them are limited to this area.

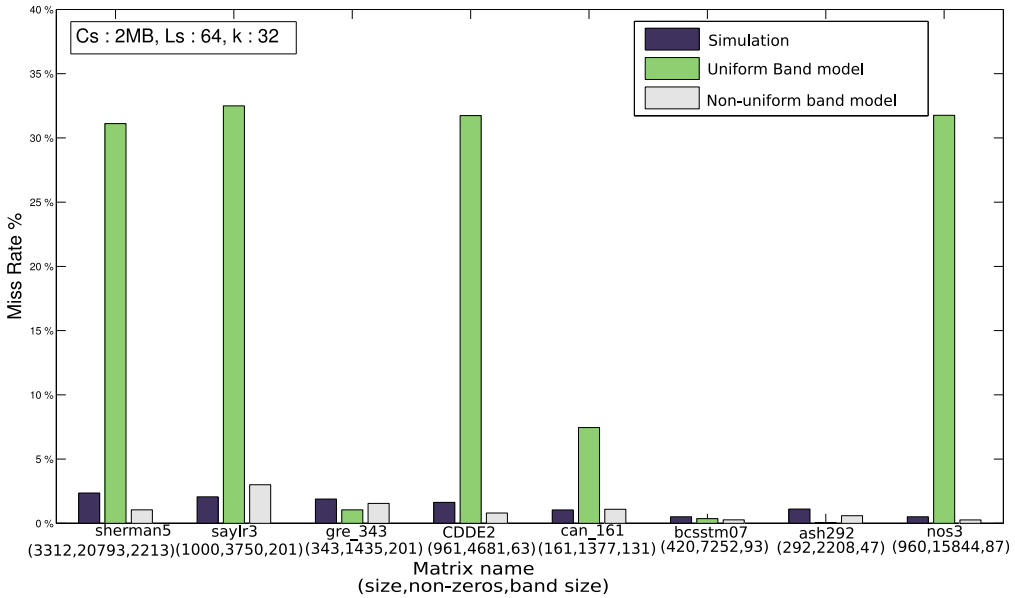
Table 4.7 summarizes data giving an idea of the accuracy of the model. The results were obtained for the benchmarks performing 1770 tests considering 10 different cache configurations of each one of the 177 matrices of the Harwell-Boeing and the NEP sets. For each matrix and cache configuration 10 different simulations were performed changing the base address of the data structures involved in each code. In the case of the three orderings of the sparse-matrix dense-matrix product the number of columns of the dense matrix is always a half of its number of rows. The cache configurations have cache sizes (C_{s}) from 16 KBytes to 2 MBytes, line sizes (L_{s}) from 16 to 64 bytes and associativity degrees (K) 1, 2, 4 and 8. Column $\overline{MR_{\text{Sim}}}$ contains the average value of the miss rate simulated in the set of experiments. Column $\overline{\sigma_{\text{Sim}}}$ is the average typical deviation of the miss rate obtained in the 10 simulations performed changing the base address of the data structures. The table compares the precision of the predictions achieved using the simple model for banded matrices assuming an uniform distribution of nonzeros introduced in Section 4.2 and the improved model presented in this paper. The table shows for each model, $\overline{MR_{\text{Mod}}}$ the average value of the miss rated predicted, and $\overline{\Delta_{MR}}$ the average value of the absolute value Δ_{MR} of the difference between the predicted and the measured miss rates for each experiment. We use absolute values, so that negative

errors are not compensated with positive errors. These results show that the improved model is much mode accurate in the presence of real heterogeneous input banded matrices than the original model. The small values of $\overline{\sigma_{\text{Sim}}}$ point out that the base addresses of the data structures play a minor role in the cache behavior.

Figure 4.14 contains a comparison of the miss rate obtained in the simulation, the miss rate obtained by the uniform bands model and the miss rate obtained by the non-uniform bands model during the execution of the sparse matrix-dense matrix product with IJK ordering using some matrices from the Harwell-Boeing and the NEP collections. The number of columns of the dense matrix used in the multiplication was always one half of the number of rows of the sparse matrix. Figure 4.14(a) shows the results obtained using a typical level 1 cache configuration, while a typical level 2 cache configuration is used in Figure 4.14(b). The cache configuration parameters are: C_{s} the total cache size, L_{s} the line size and K the degree of associativity. The non-uniform bands model almost always estimates more accurately the miss rate. The difference is bigger in the level 2 cache configuration. The reason for the poor estimations obtained using the uniform bands model is that in matrices with wide bands but in which most of the values are concentrated in a few diagonals, there is a lot of reuse that is not captured by the uniform bands model, as it assumes that the entries are uniformly spread along all the diagonals in the band. The predictions for matrices such as sherman5, gre343 and ash292 are less accurate because they do not fit exactly in the form described in 4.12, as in some of their diagonals the density of the nonzeros is not uniform, that is, some diagonals exhibit different densities along their length. The predictions for the level 1 cache configurations using the uniforms band model are sometimes relatively accurate. The reason is that although this model often mispredicts the reuse distance for the accesses with an irregular access pattern, the associated miss probability is so high in this cache for some matrices even for short reuse distances that this error does not affect the accuracy of the prediction as much as in the case of a bigger cache like the typical level 2 cache configuration.



(a) Simulation and modeling for a typical level 1 cache configuration



(b) Simulation and modeling for a typical level 2 cache configuration

Figure 4.14: Comparison of the miss rates obtained by the simulation, the uniform bands model and the non-uniform bands model during the execution of the sparse matrix-dense matrix product with IJK ordering for several real matrices.

Chapter 5

Automated Implementation in a Compiler Framework

The original PME model was only automatable for codes with regular access patterns. In the previous chapters, automatable extensions of the PME model were proposed to cover irregular codes due to both data-dependent conditional statements (described in Chapter 3) and indirections (described in Chapter 4).

A fully automatic tool was built using the ideas of the original PME model capable of predicting the cache behavior in regular codes [31]. This chapter describes the full automation of the PME model extension for irregular codes due to indirections and a uniform distribution of the values. The information retrieval is harder to perform in irregular codes than in regular codes. For this purpose, we use the XARK compiler [15], an extensible framework for automatic kernel recognition that can be used as a powerful and efficient information-gathering tool [16, 17]. In order to characterize the access patterns followed by the references in the codes, a subset of the well-known chains-of-recurrences formalism was implemented in the compiler.

Section 5.1 presents a motivating example that will be used throughout this chapter. Section 5.2 introduces chains of recurrences for the characterization of the access patterns. Section 5.3 describes the algorithm to build the PME model automatically from the point of view of the information to be retrieved by the XARK compiler. Section 5.4 provides and overview of the internals of the XARK compiler and presents an extension of XARK that retrieves the information required by the model. Finally, Section 5.5 shows an example of how the automated PME model can be used to guide an optimization process succesfully.

5.1. Motivating Example

The development of the PME model extensions for irregular codes has been driven by a set of well-known codes that contain regular and irregular access patterns. A manual analysis of such codes revealed that the automation of the model from scratch is a difficult task, specially in the scope of irregular applications, as advanced symbolic analysis is needed to retrieve the necessary information.

For illustrative purposes, consider the code of Figure 4.5 for the computation of the product of a MxN sparse matrix in CRS format [19] and a NxH dense matrix. The outermost loop $\texttt{do}_\texttt{I}$ presents array references with regular access patterns. The expressions used for their indexing can be rewritten as affine functions of the enclosing loop indices. For instance, the subscript of $\texttt{R}(\texttt{I}+1)$ takes increasing values in the interval $[2,\texttt{M}+1]$. Current commercial and research compilers can gather this information. However, irregular access patterns due to indirections require advanced symbolic analysis techniques. For example, reference $\texttt{B}(\texttt{REG1},\texttt{J})$ follows an irregular access pattern because the values of REG1 are determined by $\texttt{C}(\texttt{K})$, whose values are not known at compile-time. Note that K introduces a higher level of indirection because it takes values in the interval $[\texttt{R}(\texttt{I}),\texttt{R}(\texttt{I}+1)-1]$ in each $\texttt{do}_\texttt{I}$ iteration. Further analysis of the headers of $\texttt{do}_\texttt{I}$ and $\texttt{do}_\texttt{K}$ reveals that the code traverses the whole array of row indices of the sparse CRS matrix. The recognition of this programming construct, usually referred in the literature as *offset and length* [39], leads to conclude that K takes a strictly monotonically increasing set of values during the execution of $\texttt{do}_\texttt{I}$ and, thus, different elements of array C are referenced at run-time. The accuracy of the model would increase if the compiler could retrieve this information. The XARK compiler represents access patterns by means of the chains-of-recurrences formalism, which will be introduced in Section 5.2. From these chains of recurrences the PME model will build the equations that characterize the cache behavior for such access patterns. The corresponding algorithm will be described at high level in Section 5.3. The details about the recognition of programming constructs such as offset and length will be presented in Section 5.4.

5.2. Chains of Recurrences

Chains of recurrences (CR) is a formalism to represent closed-form functions [59] that is used in different computer algebra systems, optimizing compilers and stand-

alone libraries. Chains of recurrences have been successfully used to expedite function evaluation at a number of points in a regular interval. Given a constant ϕ_0, a function g defined over the natural numbers and zero, $N_{\text{nz}} \cup \{0\}$, and the operator $+$, a *Basic Recurrence (BR)* f, represented by the tuple $f = \{\phi_0, +, g\}$, is defined as a function over $N_{\text{nz}} \cup \{0\}$ by

$$\{\phi_0, +, g\}(i) \;=\; \phi_0 + \sum_{j=0}^{i-1} g(j) \;\text{ with }\; i \in N_{\text{nz}} \cup \{0\} \tag{5.1}$$

Example 19. For example, the loop index of $\texttt{do}_\texttt{I}$ in Figure 4.5 takes integer values in the regular interval [1,M]. The BR $f = \{1, +, 1\}$ provides a closed-form function to compute the value of I at each $\texttt{do}_\texttt{I}$ iteration and thus to determine the affine memory access pattern I of array reference R(I) ■

The algebraic properties of BR's provide rules for combining several BR's into a single BR by means of arithmetic operations Let $f = \{\phi_0, \odot, g\}$ and $g = \{\mu_0, \otimes, g_1\}$ be BR's and c be a constant. Then,

$$\{\phi_0, +, g\} \pm c \;=\; \{\phi_0 \pm c, +, g\} \tag{5.2}$$

$$\{\phi_0, +, g\} * c \;=\; \{\phi_0 * c, +, c * g\} \tag{5.3}$$

$$\{\phi_0, +, g\} + \{\mu_0, +, g_1\} \;=\; \{\phi_0 + \mu_0, +, g + g_1\} \tag{5.4}$$

$$\{\phi_0, +, g\} * \{\mu_0, +, g_1\} \;=\; \{\phi_0\mu_0, *, fg_1 + gg + gg_1\} \tag{5.5}$$

Example 20. Consider the access pattern of array reference R(I + 1). The BR of the subscript expression I + 1 is computed by applying equality (5.2) to the constant 1 and the BR $\{1, +, 1\}$ that represents the loop index I. Thus, the subscript I + 1 is represented by the BR $\{2, +, 1\}$ ■

Multidimensional Chains of Recurrences (MCR) [36] provide a formalism to describe memory access patterns of multidimensional arrays. In the following, an intuitive description of MCRs based on their interpretation is presented.

Example 21. Consider the bi-dimensional array reference D(I, J) of Figure 4.5. In the scope of $\texttt{do}_\texttt{I}$, a row-major traversal of matrix D is performed, M and H being the number of rows and columns, respectively. As both rows and columns are accessed sequentially one after another, the BR $\{1, +, 1\}$ captures the access pattern

defined by the subscript expressions I and J. However, from the point of view of the cache behavior, the description of the access pattern of the multidimensional array mapped onto a linear memory model is required. Assuming column-major storage which is the case in Fortran, the MCR $_{\mathtt{J}}\{_{\mathtt{I}}\{1,+,1\},+,\mathtt{M}\}$, composed of two nested BRs, provides such information as follows. First, the inner BR $_{\mathtt{I}}\{1,+,1\}$ is evaluated according to equation (5.1) in order to locate the beginning of row number I. Next, the outermost BR $_{\mathtt{J}}\{\mathtt{I},+,\mathtt{M}\}$ is evaluated to access the row elements stored in memory locations with stride M. Within MCRs, the subscript on the left of each BR indicates the source code variable used to evaluate the BR ■

In this work only BRs and MCRs with a constant g function are used as they enable the representation of the access patterns handled by the PME model. Note that CRs provide a powerful representation that will capture more complex cases that are expected to appear in full-scale real applications, like triangular access patterns. Besides, chains of recurrences is a well-known and widely used formalism that has an extensive research associated to it which can be used in future extensions of our work.

Figure 5.1 summarizes the information requirements of the PME model for the code of Figure 4.5. For each loop, a graph of dependence relations (represented as use-def chains) between array references and loop indices is depicted. Use-def chains starting from array references are labeled with the array dimension where the target reference appears. BRs that capture loop index values and access patterns for each dimension of each array reference are shown. When enough information is available, multidimensional arrays are also annotated with MCRs and linearized MCRs. The superscript on the right of the BRs represents an average of the number of times that the BR is evaluated. The notation ? within BRs reflects that the corresponding information cannot be determined to be a constant expression at compile-time.

5.3. Information Requirements of Extended PME Model

This section describes a high-level algorithm of the PME model as well as the information requirements of its implementation in a compiler. Section 5.3.1 focuses on the construction of the equations of the model and Section 5.3.2 on the computation of the interference regions, that is, the memory regions accessed by each given

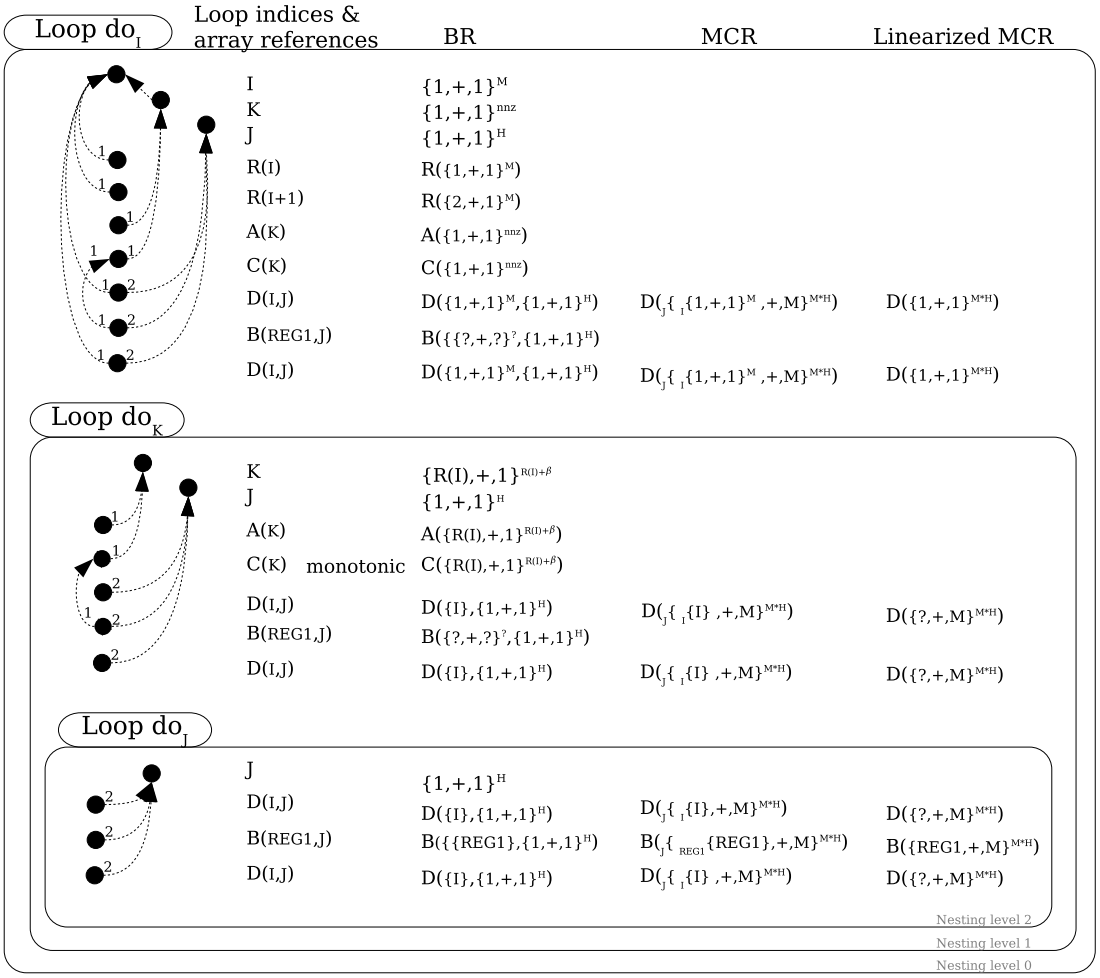


Figure 5.1: Information requirements of the PME model for the code of Figure 4.5. The symbol nnz stands for the number of nonzeros of the sparse matrix, and β is the average number of iterations of $\texttt{do}_\texttt{K}$

reference during a period of the execution of the code.

5.3.1. Constructing the Equations

The pseudo-code of Figure 5.2 gives an overview of the PME model. As shown in the top-level procedure *analyze_code*, the references that appear in each loop nest of the source code are studied one by one. Each reference R is analyzed in several scopes. At each nesting level, the procedure *number_of_misses* computes an equation that calculates the number of misses produced by that reference in that nesting level. This equation is expressed in terms of the equation of the immediately inner loop and use the function $Reg_{Ri}(n)$ defined in Figure 4.2 for calculating the interference region. A reference may exhibit different access patterns with respect to different loops. These access patterns are modeled by the following equations: the *regular access PME* for regular patterns, the *monotonic irregular access PME* for irregular patterns that access a monotonic sequence of memory positions, and the *non-monotonic irregular access PME* for irregular patterns that cannot be predicted at compile-time. Procedure *number_of_misses* selects the appropriate equation by analyzing the BRs associated with each dimension of `R` as follows:

- The regular access PME is applied if the BR matches $\{\phi_0, +, g\}$ with constant function g.
- The monotonic irregular access PME is applied if (1) a BR characterizing one of the dimensions has a non-constant g, and (2) there is a path of use-def chains between `R` and the loop index of the current loop that contains at least another different array reference. The first step of this path must be a use-def chain with a target array reference whose values can be determined to be monotonic.
- Otherwise, the non-monotonic irregular access PME is selected.

Example 22. As an example, consider the array reference `B(REG1, J)` in Figure 4.5. In the analysis of the innermost loop $\texttt{do}_\texttt{J}$, the BRs that describe every dimension of the reference are explored. As shown in Figure 5.1, the BR $\{REG1\}$, simplified representation of $\{REG1, +, 0\}$, that describes the access pattern in the first dimension, is an invariant BR. In addition, as the BR $\{1, +, 1\}$ associated with the second dimension has a constant function $g = 1$, the subscript is known to be an affine

```
procedure analyze_code() {
1    foreach loop_nest of the code {
2       foreach reference in the loop_nest {
3          misses+ = number_of_misses(reference, outermost_loop(loop_nest), R_full)
4       }
5    }
}
procedure number_of_misses(reference, loop, region) {
1   if is_regular(reference, loop) {
2      return regular_access_PME(reference, loop, region)
3   } else {
4      if is_monotonic(reference, loop) {
5         return irregular_monotonic_access_PME(reference, loop, region)
6      } else {
7         return irregular_nonmonotonic_access_PME(reference, loop, region)
8      }
9   }
}
procedure irregular_monotonic_access_PME(reference, loop, region) {
1   if is_innermost_loop_containing(loop, reference) {
2      return L_Ri * cache_impact_quantification(region) + (Ni - L_Ri) * cache_impact_quantification(Reg_Rloop(1))
3   } else {
4      misses=0.0
5       foreach inner_loop in inner_loops_containing(loop, reference){
6         misses+ = L_Ri * number_of_misses(reference, inner_loop, region)
7                 +(Ni - L_Ri) * number_of_misses(reference, inner_loop, Reg_Rloop(1))
8       }
9      return misses
10   }
}
```

Figure 5.2: The PME model algorithm

function of J. Thus, a regular access PME models the behavior of the reference in this loop.

A different situation arises in the scope of $\texttt{do}_\texttt{K}$ at nesting level one. The BR for the first dimension has unknown ϕ_0 and g, which is represented as $\{?, +, ?\}$ in Figure 5.1. Besides, the graph of dependence relations depicted in Figure 5.1 shows that there is a path from the first dimension of B(REG1, J) to loop index K that contains another array reference C(K) whose values are stored in the scalar REG1 (see lines 2, 4 and 6 of Figure 4.5). Thus, the subscript REG1 is known to be irregular. The accuracy of the prediction can be raised by taking advantage of the knowledge that C is the column array of a sparse CRS matrix since, assuming that the column indices are ordered within each matrix row, the sequence of values of C(K) is known to be monotonic. As a result, the monotonic irregular access PME is applied. Note that such information is not available in the scope of the outermost loop because C(K) is not monotonic across different iterations of $\texttt{do}_\texttt{I}$. In this case, the non-monotonic irregular access PME is used ■

Two parameters are required to build a PME at nesting level i: N_i, the number of iterations of the loop, and S_{Ri}, the stride between the elements that reference R accesses in two consecutive loop iterations. In the case of regular accesses and monotonic irregular accesses L_{Ri}, the number of loop iterations for which R cannot exploit any reuse, must be calculated too. In our algorithm, N_i is the upper bound of the BR that characterizes the values of the loop index. As for S_{Ri}, if there is not any dependence path between the reference and the loop index, $S_{Ri} = 0$. Otherwise, it is calculated as the product of the constant g of the BR associated with the loop index by the distance between two consecutive elements of the array referenced by R in the dimension indexed by the loop index. This latter value is calculated using the dimensions of the indexed array and the mapping of the array into the linear memory model (i.e., row-major or column-major). Finally, L_{Ri} is calculated using Equation 2.13 when the access is regular or using Equation 4.2 when the access is irregular monotonic.

Example 23. In regular codes, N_i is sometimes available at compile time, and thus the upper bound of the BR of the loop index can be computed (see the BR $\{1, +, 1\}^H$ of $\texttt{do}_\texttt{J}$ in Figure 5.1). However, this is not a very common situation in irregular codes. Consider the loop index K of the offset and length construct of Figure 4.5. In the scope of $\texttt{do}_\texttt{I}$, K is used in A(K) and C(K) to access the whole sparse CRS matrix. Thus, N_i is the number of nonzeros nnz, as shown in the

BR $\{1,+,1\}^{nnz}$ of Figure 5.1. In contrast, in the scope of $\texttt{do}_\texttt{K}$, N_i is given by the symbolic expression $R(I+1)-R(I)$. In general, this expression takes a different value in each iteration of the outer loop $\texttt{do}_\texttt{I}$. However, from a statistical point of view, $N_i = \beta = \frac{nnz}{M}$ can be a good approximation for CRS sparse matrices with a uniform distribution of the entries, M being the number of rows of the sparse matrix. Thus, as $\texttt{do}_\texttt{K}$ traverses the elements of a row of the CRS matrix, the values taken by K could be represented by $\{R(I),+,1\}^{R(I)+\beta}$. This situation also affects the calculation of the stride for the array reference $\texttt{C(K)}$ in the scope of $\texttt{do}_\texttt{I}$. Loop index I indexes C(K) through its dependence with the loop index K. As a result, the stride of C(K) with respect to loop $\texttt{do}_\texttt{I}$ will be the number of iterations of $\texttt{do}_\texttt{K}$ (i.e., β), because both loops define an offset and length construct ■

5.3.2. Computing the Interference Regions

In Section 2.3, three steps were described to estimate the miss probability associated to a given reuse distance: access pattern identification, cache impact quantification and area vectors addition. The cache impact quantification step uses the results generated by the access pattern identification step, which in its turn retrieves information directly from the source code. As explained in Section 4.2.1, the access pattern identification step for codes with indirections generates as intermediate representation of the access pattern of each reference R a D_A-tuple $\mathcal{R}_R(h,n)$, where D_A is the number of dimensions of the array A referenced by R. Each element of this tuple consists in its turn of a 3-tuple $\mathcal{R}_{Rj}(h,n) = (M_j, S_j, P_j)$, where the M_j is the number of different points accessed along dimension j, S_j the constant stride between two consecutive points and P_j the probability each one of these points is actually accessed by R.

The information supplied by the BRs, the MCRs and the use-def-chains represented in Figure 5.1 is used to generate this D_A-tuple $\mathcal{R}_R(h,n)$. The first step to build $\mathcal{R}_{Rj}(h,n)$ is to determine whether the indexing of dimension j is done across an indirection. This is the case if the use-def-chain path between the j-th dimension of this reference and the loop index on which it depends includes another array reference. If the indexing of the studied dimension is not done across an indirection, then the access is regular and the algorithm followed to calculate the j-th component of $\mathcal{R}_R(h,n)$ is the one described in Section 2.3. In this case the index of the reference is an affine function $\alpha_{Rj}\cdot\texttt{I}_i + \delta_{Rj}$ of some loop index $\texttt{I}_i$. The identity of $\texttt{I}_i$ can be found out exploring the use-def-chain paths. The set of points accessed in this di-

mension by R can be represented as the tuple $(\text{Iters}_i(h,n), S_{Ri}, 1.0)$, where for the calculation of $\text{Iters}_i(h,n)$ (see Sections 2.3.1, 4.2.1) the only additional information we need to extract from the code is N_i, the number of iterations of nesting level i. As for S_{Ri}, it is the stride that reference R has with respect to loop i. As in this case the index of dimension j is an affine function of $\mathtt{I}_i$, then $S_{Ri} = \alpha_{Rj} \cdot d_{\mathrm{A}j}$, where $d_{\mathrm{A}j}$ is the accumulative size of the j-th dimension of the array `A` referenced by R and α_{Rj} is the scalar that multiplies the loop variable in the affine function. $d_{\mathrm{A}j}$ can be calculated in this case using the sizes of the different dimensions of the array `A` while α_{Rj} can be extracted from the expression that indexes to the j-th dimension of the reference R.

When the indexing of dimension j depends on an indirection, that is, the index has a form $\alpha_{Rj}\cdot\mathtt{B}(f(\mathtt{I}_i))+\delta_{Rj}$, we assume that the accesses are spread uniformly on the indexed dimension of the array. The identity of the index array `B` can be obtained using the use-def-chain path, and the information about the value of α_{Rj} can be retrieved in the same way as in the regular case. The values of M_j,S_j and P_j can be calculated using the equations described in Section 4.2.1. Once the D_{A}-tuple $\mathcal{R}_R(h,n)$ has been generated and simplified, the rules described in that section are used to identify the kind of region associated to the accesses of that reference.

The information supplied by the BRs and MCRs can help us perform the access pattern identification step easier. Most of the accesses to data structures of one or two dimensions generate a small set of possible BRs and MCRs that can be easily identified and translated to their corresponding regions. For this purpose, we have developed the following rules.

- Let $\{\phi_0, +, g\}^{\Gamma}$ be the BR of a unidimensional array reference. If $g = 1$, a region $R_s(\Gamma)$ is computed. Otherwise a region $R_r(\frac{\Gamma}{g}, 1, g)$ is associated with the array reference.

- In the case of multi-dimensional arrays, the analysis focuses on the MCR that represents the access pattern once the array has been mapped onto the linear memory model. For the sake of the explanation, consider the MCR $\{\{\phi_1, +, g_1\}^{\Gamma_1}, +, g_2\}^{\Gamma_2}$ of a bi-dimensional array reference, where ϕ_1, g_1 and Γ_1 are associated with the first array dimension, and g_2 and Γ_2 with the second dimension. In this case, a region $R_r(\frac{\Gamma_2}{g_2}, \frac{\Gamma_1}{g_1}, g_2)$ is computed. Sometimes a simplified representation of the access pattern described by the MCR can be obtained by linearizing the MCR. The resulting BR is processed as described for unidimensional arrays.

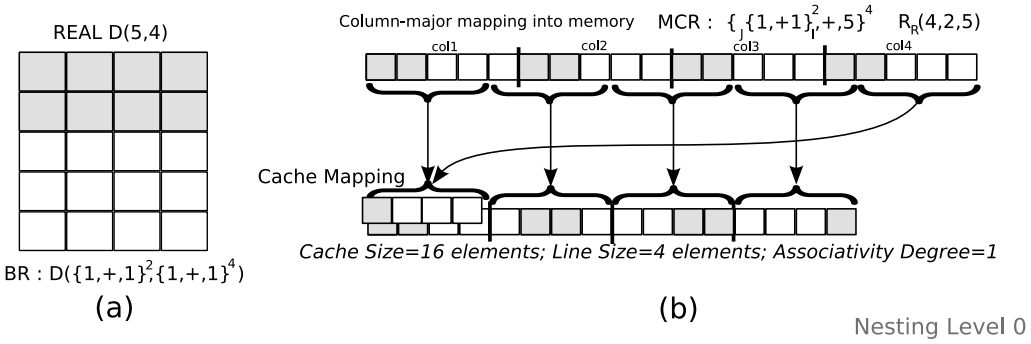


Figure 5.3: Matrix mapping in memory and in cache for reference D(I, J) of Figure 4.5 during 2 iterations of loop $\texttt{do}_\texttt{I}$

Whenever all the accesses affect consecutive memory positions the mapping and the traversal of the array are equal, the MCR can be simplified to the BR $\{1,+,1\}^{limit_1 * limit_2}$.

Once the memory region accessed by a given reference is identified, the impact quantification step estimates numerically the cache impact of the access to this region in the cache.

Example 24. In the example code of Figure 4.5, during the analysis of the reference D(I, J) in the scope of the loop $\texttt{do}_\texttt{J}$, the BR for the first dimension {I} indicates that the index is a loop invariant, while that of the second dimension $\{1,+,1\}^{\texttt{H}}$ shows that the subscript J takes consecutive values in the regular interval [1, H]. As shown in Figure 5.1, the MCR $_\texttt{J}\{_\texttt{I}\{\texttt{I}\},+,\texttt{M}\}^{\texttt{M*H}}$ of D(I, J) can be linearized as the BR $\{?,+,M\}^{M*H}$, the unknown ϕ_0 indicating that $\texttt{do}_\texttt{J}$ is analyzed in the scope of an undetermined $\texttt{do}_\texttt{I}$ iteration. Applying the rule of unidimensional array references, the memory region $R_r(H,1,M)$ of a row of array D is computed. When the access pattern for D(I, J) is analyzed in the next outer loop $\texttt{do}_\texttt{K}$, at nesting level 1, the BRs and MCRs for both dimensions are the same ones as in the innermost loop $\texttt{do}_\texttt{J}$ because none of the dimensions depends on loop index K. Thus, the same region $R_r(H,1,M)$ is computed. A different situation arises in the scope of the outermost loop, where there is a different BR $\{1,+,1\}^{\texttt{M}}$ for the first dimension. As the M rows of the matrix are accessed, the linearized MCR $\{1,+,1\}^{\texttt{M*H}}$ contains a $\phi_0 = 1$ that reflects the access to the whole array D resulting in a region $R_s(M \times H)$. Figure 5.3 shows how an access to the array D during two iterations of the outermost loop is mapped into the memory and the cache. ■

5.4. XARK Extension for the PME Model Automation

The automation of the PME model is addressed using the XARK compiler [17, 15], an extensible framework for the automatic recognition of programming constructs that are frequently used by software developers (from now on, computational kernels). It was originally developed to detect parallel loops in irregular codes, where array references with subscripted subscripts reduce the effectiveness of most dependence analyzers. Using the information provided by the kernel recognition engine, XARK was extended to provide a powerful information-gathering tool that supports the implementation of parallelizing code transformations [16]. XARK operates on top of a high-level intermediate representation resembling the original source code that consists of the forest of abstract syntax trees (ASTs) that represent the statements of the Gated Single Assignment (GSA) form of the code. GSA is an extension of the well-known Static Single Assignment (SSA) form where reaching definition information is represented syntactically. Unlike SSA, GSA captures the flow of values in a program for both scalar and array variables. In addition, the intermediate representation contains predicates that capture the conditions of if-endif constructs. These properties enable to implement the recognition engine efficiently, and widen the collection of computational kernels that can be recognized by compiler. In an AST, a tree represents an operation so that the root node is the operator (e.g., assignment, scalar fetch, array reference, plus, product) and its children are the operands. The intermediate representation is completed with use-def chains that exhibit the dependences between the statements of the code.

XARK performs a demand-driven analysis that proceeds as follows. A post-order traversal is carried out on each AST. At each node, a transfer function that gathers information about each operator in the program is applied once the analysis of the children subtrees has finished. When an occurrence of a variable defined in a different AST is found, the post-order traversal is stopped until the analysis of the latter AST is completed. This demand-driven behavior assures that all the information needed at a given node has been computed before the transfer function is actually executed.

Transfer functions are organized in layers devoted to specific tasks. The bottom layer addresses the recognition of the kernels computed in the source code (e.g., generalized induction variables, irregular reductions, array recurrences), which includes the characterization of the regular and irregular access patterns of the array

references that appear in the source code. Upper layers implement extensions of the XARK compiler that benefit from the information recognized in the source code. Information interchange between layers is carried out by means of three containers that are available in all the transfer functions: *pgm* holds information at the program unit level; *stm* at the statement level; and *node* in the scope of a node of the AST of a statement. The pseudo-code of the extension that builds the interface between XARK and the PME model is shown in Figure 5.4. Due to space limitations, the details about the computation of the BRs and the MCRs has been omitted from the transfer functions. The containers are represented as data structures whose fields correspond to pieces of information retrieved from the source code.

In order to illustrate the operation of XARK, consider the forest of ASTs and the use-def chains (dashed arrows) depicted in Figure 5.5. The details about the GSA form have been omitted for the sake of clarity. The picture shows the last step of the post-order traversal of the AST that represents the loop header `DO K=R(I),R(I+1)-1`. Hatched nodes highlight expressions and statements whose analysis has already been completed. When transfer function T_{do} is applied, the kernel recognition layer characterizes `R(I)` and `R(I+1)-1` as loop-variant expressions whose value is not known at compile-time. This is denoted by the annotation `subscripted` in the corresponding nodes of the AST. Expressions corresponding to invariant and linear access patterns are annotated as `invariant` and `linear`, respectively. To each node it is also attached the BR that captures the interval in which the expression takes values, which is computed by applying the rules defined in the CR algebra [59] (see the example in Section 5.2). Next, the extension of T_{do} presented in Figure 5.4 is executed. First, the loop header `DO K=R(I),R(I+1)-1` is recognized as an offset and length construct because `R(I)` and `R(I+1)-1` are subscripted accesses to consecutive elements of a unique array `R`, and each expression is the source of a use-def chain whose target is the outermost loop $\texttt{do}_\texttt{I}$. Under these conditions, T_{do} rewrites the BR $\{\texttt{R(I)}, +, 1\}^{\texttt{R(I+1)-1}}$ as $\{1, +, 1\}^{\texttt{nnz}}$ to indicate that the loop index `K` traverses the whole sparse matrix during the execution of $\texttt{do}_\texttt{I}$ (see lines 2 to 6 of procedure T_{do} in Figure 5.4). The demand-driven analysis of the forest of ASTs continues, and the access pattern of array reference `C(K)` is characterized as a linear pattern given by the BR $\{1, +, 1\}^{\texttt{nnz}}$.

```
struct {...                          struct {...                          struct {...
      graph_of_array_refs;                 set_of_array_refs;                   set_of_array_refs;
} pgm;                               } stm;                               } node;

 procedure T_{a(s_1,...,s_n)} {          // Extensions of transfer function of array references
1    insert a(s_1, ..., s_n) in pgm.graph_of_array_refs
2    foreach s_i with subscripted access pattern {
3       foreach reference ∈ node_{s_i}.set_of_array_refs {
4          insert a use-def chain from a(s_1, ..., s_n) to reference in pgm.graph_of_array_refs
5       }
6    }
7    insert a(s_1, ..., s_n) in node.set_of_array_refs
 }

 procedure T_x {             // Extensions of transfer function of identifiers
1    if x is not invariant {
2       foreach reference ∈ set_of_array_refs of the definition statement of x {
3          insert reference in node.set_of_array_refs
4       }
5    }
 }

 procedure T_do {          // Extensions of transfer function of loop headers
1    stm.set_of_array_refs = node_init.set_of_array_refs ∪ node_limit.set_of_array_refs ∪ node_step.set_of_array_refs
2    if stm is an offset and length construct {
3       if stm at nesting level 1 {
4          rewrite symbolic BR {R(I), +, 1}^{R(I+1)-1} as {1, +, 1}^{nnz}
5       }
6    }
 }

 procedure T_stm {             // Extensions of transfer function of assignment statements
1    stm.set_of_array_refs = node_rhs.set_of_array_refs
 }
```

Figure 5.4: Extension of XARK for building the interface with the PME model

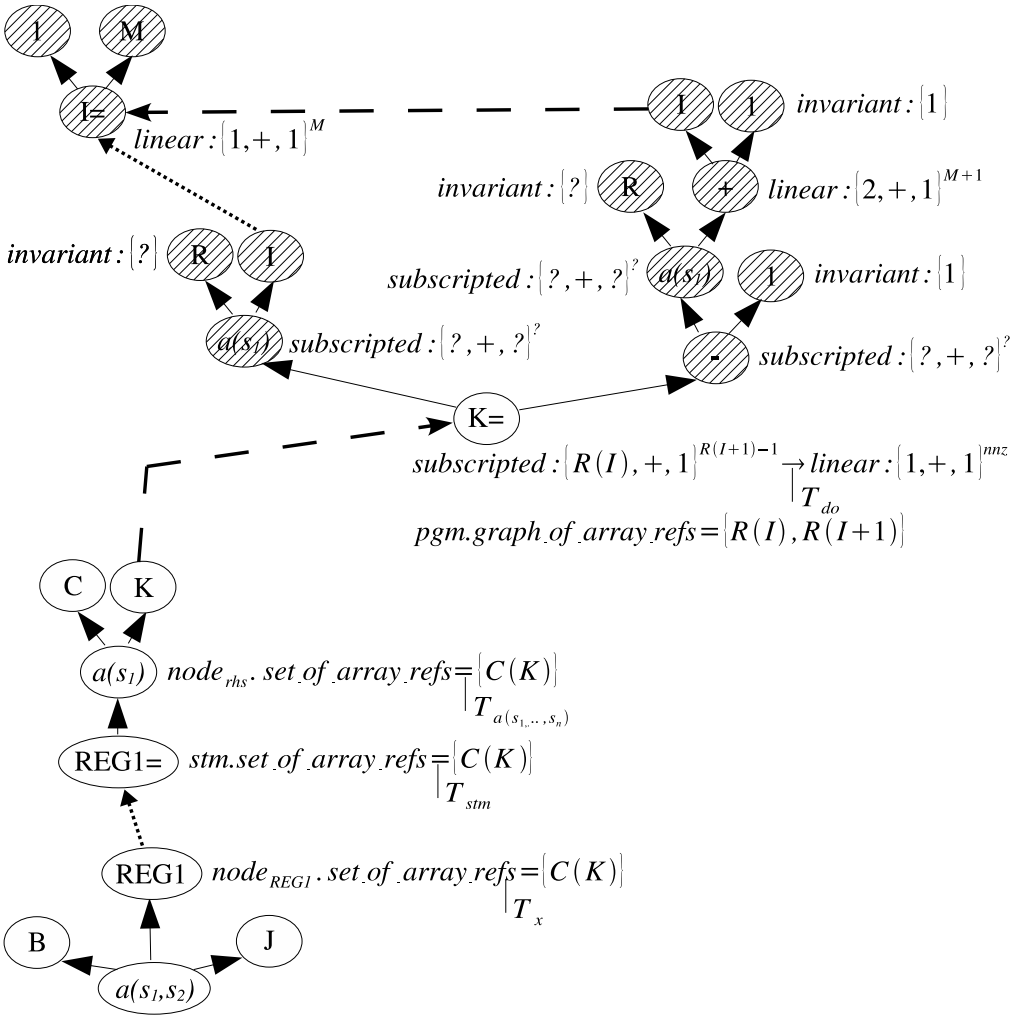


Figure 5.5: Forest of ASTs and use-def chains of the offset and length construct and the array reference B(REG1, J) of the example code of Figure 4.5.

5.4.1. Construction of the Graph of References

Apart from the characterization of the access patterns of array references, the interface between XARK and the PME model exhibits the dependence relationships between array references and loop indices. As shown in Section 5.3, such information is used to build the equations that capture the cache behavior of the source code. The graph of dependences is built as follows. Each time the transfer function of array references $T_{a(s_1,...,s_n)}$ is executed, the corresponding array reference is inserted in $pgm.graph\_of\_array\_refs$ (see line 1 of procedure $T_{a(s_1,...,s_n)}$ in Figure 5.4). Thus, when T_{do} is applied in Figure 5.5, $pgm.graph\_of\_array\_refs$ is $\{\mathtt{R(I)}, \mathtt{R(I+1)}\}$. As a result, a list of all array references in the source code has been built.

In order to construct the graph, it is necessary to identify indirections as well as the array references that appear in subscript expressions. This task is accomplished by taking advantage of the access pattern characterization provided by the kernel recognition layer.

The code is analyzed trying to recognize syntactical variation of a set of sparse computational kernels that are frequently used in full-scale applications, for instance, operations with sparse vectors and matrices. This recognition is performed by taking into account the semantics of the program. The compiler must detect occurrences in the code of different kernel types as: induction variables, scalar reduction operations, linked list-traversal and masked and array operations. These detections techniques suffer two main problems: source quality and difficult to analyze complex control constructs. The XARK compiler uses an extension of the classification scheme of the technique proposed by Gerlek, Stoltz and Wolfe [40] capable of recognizing complex induction variables even in loops that have a complicated control flow. Induction variables can be substituted by closed form expressions. This extension can deal with both scalar values and arrays.

The demand-driven nature of XARK assures that the access pattern of each subscript s_l $(1 \leq l \leq n)$ has been characterized before the transfer function is applied. Thus, $T_{a(s_1,...,s_n)}$ recognizes array references that are not indirections by checking that there is not any subscripted access pattern, and inserts the reference in the container available for each node of the ASTs, in particular, in $node.set\_of\_array\_refs$ (line 7 of $T_{a(s_1,...,s_n)}$ in Figure 5.4). If an indirection is recognized, the demand-driven analysis carried out by XARK assures that $node_{s_j}.set\_of\_array\_refs$ contains the array references that appear in the subscript expression of the j$-th$ array dimension. Next, $T_{a(s_1,...,s_n)}$ inserts in $pgm.graph\_of\_array\_refs$ a set of use-def

chains whose source is $a(s_1, ..., s_n)$ and whose targets are the array references included in $node_{s_j}.set\_of\_array\_refs$ (lines 2-6 of $T_{a(s_1,...,s_n)}$ in Figure 5.4). Note that the sets of array references are transferred through scalar definition statements and loop headers. On the one hand, T_x transfers information from the container of the AST where x is defined (see lines 2-4 of T_x in Figure 5.4) to the local container node associated with the node where x is referenced. On the other hand, T_{stm} and T_{do} annotate the statements of the code with the list of array references that appear as operands of the right-hand side operators (see $node_{rhs}$, $node_{init}$, $node_{limit}$ and $node_{step}$ in Figure 5.4). As the ASTs are analyzed only once during the demand-driven analysis, the annotation of statements enables the retrieval of the set of array references for different occurrences of a scalar variable.

For illustrative purposes, consider the construction of the graph depicted in Figure 5.1 for the scope $\texttt{do}_\texttt{K}$. In particular, focus on the subscript `REG1` of the first dimension of $\texttt{B}(\texttt{REG1}, \texttt{J})$. When the AST of $\texttt{REG1} = \texttt{C}(\texttt{K})$ is analyzed, $T_{a(s_1,...,s_n)}$ inserts $\texttt{C}(\texttt{K})$ in $node_{rhs}.set\_of\_array\_refs$ and later T_{stm} annotates the statement by copying $\texttt{C}(\texttt{K})$ into $stm.set\_of\_array\_refs$. Next, the occurrence `REG1` in $\texttt{B}(\texttt{REG1}, \texttt{J})$ is processed by T_x, which obtains $\texttt{C}(\texttt{K})$ from the AST container of the statement where `REG1` is defined. As a result, the array reference $\texttt{C}(\texttt{K})$ is available at T_x, which copies $\texttt{C}(\texttt{K})$ in the local container $node_{REG1}.set\_of\_array\_refs$ to expose such information to $T_{a(s_1,...,s_n)}$. Finally, $T_{a(s_1,...,s_n)}$ updates the global container $pgm.graph\_of\_array\_refs$ with a use-def chain from $\texttt{B}(\texttt{REG1}, \texttt{J})$ to $\texttt{C}(\texttt{K})$.

5.5. Experimental Results

The accuracy of the automated PME model for codes with indirections integrated in the XARK compiler was widely proved with the experiments described in Section 4.2.4 (see Table 4.1). The ability to apply automatically the PME model to a wide range of codes, and its high degree of accuracy make it a powerful tool to guide compiler optimizations.

5.5.1. Driving compiler optimizations

Analytical models can be used to provide insights about the cache memory behavior of codes and can guide optimizations in a compiler or interactive tool based on their predictions. Namely, decisions can be taken based on a cost function that

Architecture	L1 Parameters $(C_{s_1}, L_{s_1}, K_1, W_1)$	L2 Parameters $(C_{s_2}, L_{s_2}, K_2, W_2)$	L3 Parameters $(C_{s_3}, L_{s_3}, K_3, W_3)$
Itanium 2	(16K,64,4,8)	(256K,128,8,24)	(6MB,128,24,120)
PowerPC 7447A	(32K,32,8,9)	(512K,64,8,150)	-

Table 5.1: Memory hierarchy parameters in the architectures used (sizes in bytes), miss weights W in CPU cycles.

considers the relative costs of the misses in each memory level as well as the CPU cycles. Memory stall time can be estimated by applying the model to the different levels of the memory hierarchy of the computer simultaneously and multiplying the number of misses estimated for each level by its miss penalty. The cycles spent in the CPU can be estimated using CPU models such as Delphi [24], which can apply heuristics to account for the properties of current high ILP superscalars. Several papers in the bibliography illustrate the success of this approach for different optimizations such as padding [52] or tiling [51, 29] in codes with regular access patterns.

As a simple experiment aimed to prove that our model can be used to optimize codes with irregular access patterns due to indirections, we used its predictions to decide which was the best loop ordering for the sparse matrix-dense matrix product using two very different architectures and memory hierarchies: Itanium 2 at 1.5GHz and a PowerPC 7447A at 1.5GHz. Table 5.1 shows the configuration of their memory hierarchies using the well-known notation C_s, L_s and K, using bytes to measure sizes. A new parameter W, the cost in CPU cycles of a miss in the considered memory hierarchy level, is also taken into account. Notice that the first level cache of the Itanium 2 does not store floating point data; so it is only used for the study of the behavior of the references to arrays of integers. Also, the PowerPC does not have a third level cache.

Our model predicted the same behavior in both architectures for every sparse matrix: the JIK ordering would be the one that would give place to the best performance, while IKJ would be the ordering that would generate more misses in all the levels of the memory hierarchy, thus yielding the worst performance. This matches the global results displayed in Table 4.1. The predictions were first validated executing the three versions of the sparse matrix-dense matrix product code for synthetic sparse matrices with an uniform distribution of the entries of sizes $N \times N$ that were multiplied by a $N \times N$ dense matrix with $N = i \times 500$ for $i = 1, 2, 3, 4, 5$ and 6, and a percentage of nonzeros in the sparse matrix from 1% to 19% in steps of 2%. The

Architecture	Synthetic uniform matrices			Real non-uniform matrices		
	Loop ordering			Loop ordering		
	IKJ	JIK	IJK	IKJ	JIK	IJK
Itanium 2	172.264	41.016	142.389	4.814	2.078	2.115
PowerPC 7447A	338.538	29.256	54.272	12.585	1.990	3.688

Table 5.2: Average execution time in seconds for the sparse matrix-dense matrix product as a function of the loop ordering.

codes were compiled using g77 3.4.3 with level of optimization `-O3`. The execution times reflected systematically the predictions of the model: the JIK version always outperformed the IJK version, being the IKJ code the slowest one. We also run a test multiplying each one of the 320 real matrices used in the preceding section by a dense matrix with 1500 columns using the three loop orderings in both machines. In the Itanium 2, the JIK ordering was the best one for 307 of the matrices, IJK for ten, and IKJ for just three of them; while in the PowerPC the JIK ordering was the fastest one in all but one of the cases, in which IJK outperformed it. Our model always chose the JIK order (see Table 4.1) in both architectures. The tests were also performed using all the banded matrices from the Harwell-Boeing and the NEP collections, in multiplications with dense matrices with 1500 columns. These tests agreed with the predictions of the model: the JIK version was the fastest one in 95.9% and 99.7% of the experiments in both architectures. Table 5.2 displays the average execution time for the three loop orderings in the two sets of experiments for both architectures in order to give an idea of the accuracy of the predictions of the model, as well as the impact of this optimization in the execution time.

Conclusions and Future Work

Conclusions

Most of the existing analytical models of the cache behavior only cover the modeling of codes with regular access patterns. The modeling of irregular codes has been only been achieved successfully for some specific kernels. The attempts to obtain an automatic approach to model the cache behavior in the presence of irregular access patterns are mainly based in heuristics and they do not obtain good degrees of accuracy. In this work, we have proposed some extensions of the PME model that cover some of the main sources of irregularity in the accesses of a code. The management of statistical information about the input data is the key idea to model this kind of codes without resorting to execute them.

We have proposed an automatable and modular extension for codes with conditional statements in which the probability that the condition is true is uniform in each one of their evaluations. The accuracy of this extension has been verified by comparing the model predictions with the results of trace-driven simulations. The model has been applied by hand to several codes of increasing complexity. Predicting the cache behavior using an analytical model is itself a very complex task even if the access patterns it presents are regular. The presence of irregular access patterns increases the difficulty associated to this task. Despite this complexity, the degree of accuracy of the predictions of our model is still high. Also, although the modeling of these codes is more demanding computationally than that of regular codes, the execution time of the model is very short, always less than one second.

We have also extended the model for codes with indirections. The main source of codes considered in this extension are those that perform sparse computations. First, we considered sparse matrices in which the non zero values of the matrix are uniformly spread along the structure. The extension proposed for this situation

obtained a good degree of accuracy in its predictions. Nevertheless, an exploration of well-known collections of sparse matrices like the Harwell Boeing and NEP collections, revealed that a high percentage of these matrices are banded, that is, most of their non zero values are spread along a limited band of the matrix. This fact led us to propose an approach that covers the modeling of this kind of matrices. First, a small modification for uniform banded matrices was proposed. Later, an additional extension was proposed for banded matrices with a non-uniform distribution of the values along the band. The accuracy of the model was verified using codes of increasing complexity that perform sparse computations by comparing the predictions of the model with the results of trace-driven simulations. The predictions of the model were very accurate despite the short time required to evaluate it. Namely, it provides its predictions always in less than one second, even for the cases in which the execution of the analyzed code takes several minutes.

The next step was to implement the effective automation of one of these extensions. Specifically, the one for codes with indirections and an uniform distribution of the values was chosen. For this purpose we used an advanced compilation framework, the XARK compiler. This compiler can extract the information needed by the model from the source code of the analyzed program. The automation of the whole process allowed us to model both the codes used in the manual validation of this extension and new codes from the SPARSKIT library. The results show that the predictions are the same as those obtained when the model was applied by hand. Up to this point, the time required to apply the model did not include the time needed to derive the formulas, as these ones were derived by hand. Once this task can be performed automatically, the time necessary to execute the whole modeling process is still short, and in many cases several orders of magnitude shorter than the time necessary to execute the analyzed code.

One of the main applications of this kind of models is to help guide optimization processes. Thus, we performed an experiment in which we used the PME model to select the optimal nesting order for the sparse matrix-dense matrix product. Several tests were performed considering different architectures with different cache configurations and changing the densities and matrices sizes. The selection of the PME model always matched the best order according to the timing of the execution of the analyzed codes in the corresponding architectures. These tests were performed using both synthetic and real matrices. This experiment shows that although the quantitative estimation performed by the model for real matrices (with a non-uniform distribution) is not very accurate, it predictions can be used successfully to guide

an optimization process.

The extension of the scope of application of the PME model to irregular codes is a big step forward in the effective utilization of analytical modeling as a method to predict the cache behavior instead of traditional techniques like trace-driven simulation and hardware counters. This research is of great interest, since irregular codes usually lack locality and thus their performance can be improved by increasing their locality guided by models like the one we have developed. These extensions keep all the desirable characteristics in a technique to study the cache behavior: accuracy, short execution time and the ability to provide insights into the reasons for the observed cache behavior

Future Work

In the future we plan to model the cache behavior of multicore architectures since they are becoming more and more common nowadays. The complexity and novelty in the analysis of the memory hierarchy of these architectures lies in the existence of several processors that can share one or several cache levels. We will try to model this situation using the PME model as a basis. In this work it has only be implemented an effective automations of the PME model extension for codes with indirections and an uniform distribution. The effective automation of the model can be improved both for codes with indirections for banded matrices, and for codes with conditional statements.

Due to its accuracy, speed and wide scope of application, this model has become a powerful tool to predict the cache behavior. We are planning to use the model to guide optimizations on both regular or irregular codes, besides those already illustrated in this thesis. The model will guide optimizations such as optimal tile size selection in the tiling technique or methods to guide the data prefetching using the model predictions. It would also be interesting to use the capabilities of the model in the field of the embedded systems and to improve their performance taking advantage of its predictions. Little work has been developed in the field of the memory behavior modeling of this kind of systems. We plan to derive estimations of the minimum and maximum number of misses in both regular and irregular codes and use them in applications such as the calculation of the WCET (Wors Case Execution Time), an open problem in embedded systems.

Bibliography

[1] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Department of Electrical Engineering, University of Stanford, 1987. pages 11, 14

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* 1986. pages 9

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* 2002. pages 9

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 85–96, 1997. pages 10

[5] D. Andrade, , M. Arenaz, B. B. Fraguela, T. J., and R. Doallo. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concurrency and Computation: Practice and Experience*, 2007. (Submitted on December 2006). pages 5

[6] D. Andrade, M. Arenaz, B. B. Fraguela, J. Touriño, and R. Doallo. Automated and accurate cache behavior analysis for codes with irregular access patterns. In *In Proceedings of Workshop on Compilers for Parallel Computers*, pages 179–193, A Coruña, Spain, January 2006. pages x, 3, 5

[7] D. Andrade, B. Fraguela, and R. Doallo. Efficient and accurate analytical modeling of the cache behavior of complete scientific codes. In *IASTED Intl. Conf. on Applied Simulation and Modelling 2003*, pages 106–111, Marbella, September 2003. pages viii, 2, 5

[8] D. Andrade, B. Fraguela, and R. Doallo. Modelado de caches ante códigos con condicionales dependientes de datos. In *Actas de las XIV Jornadas de Paralelismo*, pages 281–286, Leganés, Septiembre 2003. pages VIII, 2, 5

[9] D. Andrade, B. Fraguela, and R. Doallo. Modeling the cache behavior of codes with arbitrary data-dependent conditional structures. In Springer-Verlag, editor, *In Proceedings of the Asia-Pacific Computer Systems Architecture Conference*, volume 3189 of *Lecture Notes in Computer Science*, pages 44–57, Beijing, China, September 2004. pages VIII, 2, 5

[10] D. Andrade, B. Fraguela, and R. Doallo. Modelado analítico automático del comportamiento de la caché para códigos con indirecciones. In *Actas de las XVI Jornadas de Paralelismo*, pages 321–328, Granada, Septiembre 2005. pages IX, 2, 5

[11] D. Andrade, B. B. Fraguela, and R. Doallo. Cache behavior modeling of codes with data-dependent conditionals. In Springer-Verlag, editor, *In Proceedings of Workshop on Software and Compilers for Embedded Systems*, volume 2826 of *Lecture Notes in Computer Science*, pages 373–387, Vienna, Austria, September 2003. pages VIII, 2, 5

[12] D. Andrade, B. B. Fraguela, and R. Doallo. Analytical modeling of codes with arbitrary data-dependent conditional structures. *Journal of Systems Architecture*, 52:394–410, July 2006. pages VIII, 2, 5

[13] D. Andrade, B. B. Fraguela, and R. Doallo. Cache behavior modelling for codes involving banded matrices. In *Proc. of the 19th Intl Workshop on Languages and Compilers for Parallel Computing*, New Orleans, November 2006. pages IX, X, 2, 3, 5

[14] D. Andrade, B. B. Fraguela, and R. Doallo. Precise automatable analytical modeling of the cache behavior of codes with indirections. *ACM Transactions on Architecture and Code Optimization*, 2007. Accepted for publication. pages IX, IX, 2, 5

[15] M. Arenaz, J. Touriño, and R. Doallo. XARK: An eXtensible framework for Automatic Recognition of computational Kernels. *ACM Trans. Prog. Lang. Syst.* (Submitted on December 2006). pages 4, 74, 97, 108

[16] M. Arenaz, J. Touriño, and R. Doallo. Compiler support for parallel code generation through kernel recognition. In *18th Int. Parallel and Distributed Processing Symposium*, Santa Fe, April 2004. pages 97, 108

[17] M. Arenaz, J. Touriño, and R. Doallo. A gsa-based compiler infrastructure to extract parallelism from complex loops. In *17th ACM Int. Conf. on Supercomputing*, pages 193–204, San Francisco, June 2004. pages 97, 108

[18] Z. Bai, D. Day, J. Demmel, and J. Dongarra. A test matrix collection for non-Hermitian eigenvalue problems, release 1.0, September 1996. pages x, 83, 92

[19] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods. *SIAM Press*, 1994. pages 3, 71, 72, 85, 90, 98

[20] A. Bayona, K. London, S. Moore, P. Mucci, M. Nieto, L. Salayandia, P. Teller, and D. Terpstra. Papi deployment, evaluation, and extensions. *Proc. of the User Group Conference, 2003*, pages 349–353. pages 10

[21] D. Buck and M. Singhal. An Analytic Study of Caching in Computer Systems. *J. of Parallel and Distributed Computing*, 32(2):205–214, Feb. 1996. pages 12, 14

[22] C. Cascaval and D. Padua. Estimating cache misses and locality using stack distances. In *In ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 150–159, New York, 2003. pages 13

[23] C. Cascaval, L. D. Rose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In *Languages and Compilers for Parallel Computing*, pages 365–379, 1999. pages 13, 14, 15

[24] G. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2000. pages 13, 114

[25] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors-. In *Proceedings of the 30th Annual IEEE/ACM International Symposium Microarchitecture*, pages 292–302, December 1997. pages 10

[26] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proc. ACM Conference on Programming Languages Design and Implementation*, 2003. pages 12, 14

[27] I. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1986. pages 57

[28] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report CERFACS TR-PA-92-96, October 1992. pages x, 78, 83, 92

[29] B. B. Fraguela, M. G. Carmueja, and D. Andrade. Optimal tile size selection guided by analytical models. In *Procs. of Parallel Computing*, volume 33, pages 565–572, Malaga, Spain, September 2005. Publication Series of the John von Neumann Institute for Computing (NIC). pages 114

[30] B. B. Fraguela, R. Doallo, and E. L. Zapata. Modeling Set Associative Caches Behavior for Irregular Computations. *ACM Performance Evaluation Review (Proc. SIGMETRICS/PERFORMANCE'98)*, 26(1):192–201, June 1998. pages vii, 1, 13, 14

[31] B. B. Fraguela, R. Doallo, and E. L. Zapata. Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. *IEEE Transactions on Computers*, 52(3):321–336, March 2003. pages vii, 1, 3, 4, 13, 14, 17, 31, 32, 62, 97

[32] 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):702–745, July 1999. pages 13, 14, 15

[33] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, October 1999. pages 13, 14, 15

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 4 edition, 2006. pages 7

[35] W. King. Analysis of paging algorithms. In *Proceedings of IFIP Congress*, pages 485–490, August 1972. pages 12

[36] V. Kislenkov, V. Mitrofanov, and E. Zima. A gsa-based compiler infrastructure to extract parallelism from complex loops. In *Proc. Int. Symposium on Symbolic and Algebraic Computation*, pages 199–206, Rostock, Germany, 1998. pages 99

[37] R. E. Ladner, J. D. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proc. of the 10th annual ACM-SIAM Symposium on Discrete Algorithms (SODA99)*, pages 613–622, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. pages 13, 14, 15

[38] M. Laurenzano, B. Simon, A. Snavely, and M. Gunn. Low cost trace-driven memory simulation using simpoint. *ACM SIGARCH Computer Architecture News*, 33(5):81–86, December 2005. pages 10

[39] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 41–56, Pittsburgh, 1998. pages 90, 98

[40] E. S. Michael P. Gerlek and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 1(17):85–122, 1995. pages 112

[41] R. W. Quong. Expected I-Cache Miss Rates via the Gap Model. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 372–383, Chicago, IL, USA, Apr. 1994. IEEE Computer Society Press. pages 12, 14

[42] P. J. H. S. Chatterjee, E. Parker and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Programming Language Design and Implementation*, pages 286–297, 2001. pages 12, 14, 15

[43] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990. pages xi, 63, 73, 85

[44] I. Simecek and P. Tvrdik. Analytical model for analysis of cache behavior during cholesky. *International Conference on Parallel Processing Workshops, 2004 (ICPP 2004)*. pages 12, 14, 15

[45] A. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–53, September 1982. pages 8

[46] J. Sánchez and A. Gónzalez. Analyzing data locality in numeric applications. *IEEE Micro*, 20(4):58–66, August 2000. pages 14

[47] O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on usual numerical dense loop nests. In *Proceedings of the IEEE, special issue on Computer Performance Evaluation*, 1993. pages 12, 14, 15

[48] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proc. Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271. ACM Press, May 1994. pages 12

[49] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing*, pages 578–587, 1992. pages 13, 14

[50] R. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997. pages 9

[51] X. Vera, J. Abella, A. Gonzalez, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 68–78, New Orleans, Louisiana, October 2003. pages 114

[52] X. Vera, J. Llosa, and A. Gonzalez. Near-optimal padding for removing conflict misses. In *Proc. Languages and Compilers for Parallel Computers (LCPC02)*, volume 2481 of *Lecture Notes in Computer Science*, pages 329–343, College Park, Maryland, July 2005. LNCS - Springer Verlag. pages 114

[53] X. Vera and J. Xue. Efficient Compile-Time Analysis of Cache Behaviour for Programs with IF Statements. In *5th Int. Conf. on Algorithms and Architectures for Parallel Processing*, pages 396–407, October 2002. pages 13

[54] X. Vera and J. Xue. Let's Study Whole-Program Behaviour Analytically. In *Proc. of the 8th Int. Symposium on High-Performance Computer Architecture (HPCA8)*, pages 175–186, February 2002. pages 13

[55] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, Redwood City, 1996. pages 9

[56] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput.*, 53(5):547–566, 2004. pages 13, 14

[57] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. The future of simulation: A field of dreams? *IEEE Computer*, pages 22–29, November 2006. pages 9

[58] M. Zagha, B. Larson, and S. Turner. Performance analysis using the mips r10000 performance counters. In *Proceedings of the Supercomputing Conference*, pages 17–22, November 1996. pages 10

[59] E. Zima. Simplification and optimization of transformations of chains of recurrences. In *Proc. Int. Symposium on Symbolic and Algebraic Computation*, pages 42–50, Montreal, Canada, 1995. pages 98, 109