Softwaretechnieken ter verbetering van datalokaliteit en cachegedrag

Software Methods to Improve Data Locality and Cache Behavior

Kristof Beyls

Promotor: prof. dr. ir. E.H. D'Hollander

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Toegepaste Wetenschappen:
Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Toegepaste Wetenschappen
Academiejaar 2003–2004

Dankwoord

Het onderzoek en de resultaten beschreven in dit doctoraat zouden nooit mogelijk geweest zijn zonder de steun en de hulp van vele en verschillende personen en instanties. Het is bijna onmogelijk om niemand over het hoofd te zien, maar desalnietemin zal ik hier een poging doen om de belangrijkste personen die rechtstreeks of onrechtstreeks hebben bijgedragen aan dit werk, te vernoemen.

Prof. Erik D'Hollander, mijn promotor, die ongetwijfeld de grootste rechtstreekse bijdrage heeft geleverd. Hij was gedurende de laatste vijf jaar steeds beschikbaar om over mijn vele vragen te discussiëren. Naast de morele steun, was vooral zijn raad i.v.m. het neerschrijven en presenteren van het onderzoek in publicaties heel waardevol.

De collega's met wie ik de voorbije jaren intensief heb samengewerkt op wetenschappelijk vlak: dr. Yijun Yu, Sven Verdoolaege (Katholieke Universiteit Leuven), Rachid Seghir en prof. Vincent Loechner (Université Louis Pasteur Strasbourg). De vele discussies die ik met hen gevoerd heb, hebben mijn inzichten in de onderzoeksmaterie aangescherpt.

Prof. Chatterjee (IBM T.J. Watson Research Center) en prof. Ding (Rochester University), voor de leerrijke discussies gedurende de laatste jaren, en omdat ze speciaal uit de Verenigde Staten wouden afkomen om in de examencommissie van dit doctoraat te zetelen.

De leden van de PARIS-onderzoeksgroep, voor de aangename werksfeer en collegialiteit.

Het IWT, voor de financiering van het gevoerde onderzoek.

Mijn ouders en familie, die op onrechtstreekse wijze misschien de belangrijkste bijdrage hebben geleverd door mij sinds m'n kinderjaren het nodige doorzettingsvermogen aan te kweken.

De vele personen die in de voorbije jaren mijn leven kleur gegeven hebben buiten het domein van de computerwetenschappen. Hierbij denk ik in de eerste plaats aan Mieke, Kim, Viola, Sara, Eva, Pieter, Elle, Elke en Ine.

Kristof Beyls
Gent, juni 2004

Examencommissie

Prof. P. Kiekens, voorzitter
Decaan Faculteit Toegepaste Wetenschappen
Universiteit Gent

Prof. J. Van Campenhout, secretaris
Vakgroep ELIS, Faculteit Toegepaste Wetenschappen
Universiteit Gent

Prof. E. D'Hollander, promotor
Vakgroep ELIS, Faculteit Toegepaste Wetenschappen
Universiteit Gent

Prof. M. Bruynooghe
Departement Computerwetenschappen,
Faculteit Toegepaste Wetenschappen
Katholieke Universiteit Leuven

Prof. S. Chatterjee
IBM T.J. Watson Research Center /
University of North Carolina
USA

Prof. K. De Bosschere
Vakgroep ELIS, Faculteit Toegepaste Wetenschappen
Universiteit Gent

Dr. F. De Turck
Vakgroep INTEC, Faculteit Toegepaste Wetenschappen
Universiteit Gent

Prof. C. Ding
Computer Science Departement
University of Rochester, USA

Prof. A. Hoogewijs
Vakgroep Zuivere Wiskunde en Computeralgebra,
Faculteit Wetenschappen
Universiteit Gent

Prof. D. Stroobandt
Vakgroep ELIS, Faculteit Toegepaste Wetenschappen
Universiteit Gent

Contents

I Dutch Summary: Softwaretechnieken ter verbetering van datalokaliteit en cachegedrag **1**

1 Inleiding **3**
1.1 Achtergrond 3
1.2 Overzicht 6

2 Cache Remapping **9**
2.1 Lustegeling 9
2.2 Cachegeheugen 10
2.3 Cache Remapping vanuit Vogelperspectief 11
2.4 Cache Remapping vanuit Kikvorsperspectief 13
2.4.1 Softwarecontrole over het Cachegedrag 13
2.4.2 Draadscheduling op een Enkeldradige Processor 13
2.5 Implementatie 15
2.6 Samenvatting 15

3 De Hergebruiksafstand als Maat voor Datalokaliteit **17**
3.1 Terminologie 17
3.2 Invloed van Compileroptimalisaties op Hergebruiksafstand en Cachemissers 20
3.3 Samenvatting 22

4 Hergebruiksafstandsvergelijkingen **23**
4.1 Polyhedraal Programmamodel 24
4.1.1 Polytopen en Presburger-formules 24
4.1.2 Programmavoorstelling in het Polyhedraal Model 26
4.2 Hergebruiksafstandsvergelijkingen 27
4.2.1 Hergebruikspaar 27
4.2.2 Aangesproken Data tussen Hergebruiken 29
4.2.3 Hergebruiksafstand van een Hergebruikspaar . . 31

4.3 Enumeratie van Geparameteriseerde Polytopen 32
4.4 Samenvatting 33

5 Cache Hint-Selectie 35
5.1 Cache Hints in EPIC Architecturen 35
5.2 Statische Hint Selectie 37
5.2.1 Cache Hint per Toegang 37
5.2.2 Cache Hint per Instructie 37
5.3 Dynamisch Hint Selectie 38
5.3.1 Codegeneratie 39
5.3.2 Overheadreductie 39
5.4 Experimenten 41
5.5 Samenvatting 45

6 Visualisatie van Lange-afstandshergebruik 47
6.1 Visualisatie van Hergebruik met Lage Lokaliteit 48
6.1.1 Platformonafhankelijke Cache-optimalisaties . . . 48
6.1.2 Lokaliteitsmetrieken 49
6.1.3 Visualisatie 49
6.2 Experimenten 50
6.3 Samenvatting 53

7 Besluit 55

II Software Methods to Improve Data Locality and Cache Behavior 59

1 Introduction 61
1.1 Background 61
1.2 Overview and Contributions 65

2 Cache Remapping 69
2.1 Overview of Existing Program Optimizations 69
2.1.1 Reducing Capacity Misses 70
2.1.2 Reducing Conflict Misses 71
2.1.3 Parallel Computation to Hide Memory Latency . 72
2.1.4 Improving Replacement Decisions 72
2.2 The Cache Remapping Method 73
2.2.1 Tiled Loop Nests 73
2.2.2 Cache Memory 74

2.2.3 Conflict Misses in Tiled Algorithms 77
2.2.4 High-Level View of Cache Remapping 77
2.2.5 Low-Level Details 80
2.3 Implementation and Results 82
2.3.1 Processor Requirements 82
2.3.2 Simulation 84
2.4 Comparison with Related Work 86
2.5 Summary 87

3 The Reuse Distance Metric 89
3.1 Cache and Locality Model 90
3.1.1 Cache Terminology 90
3.1.2 Reuse Distance Terminology 91
3.1.3 Relationship with Other Locality Models 93
3.2 Fast Reuse Distance Profiling 95
3.2.1 Exact Measurement using Treaps 95
3.2.2 Approximate Measurement 97
3.2.3 Program Instrumentation 97
3.3 Effect of Compiler Optimizations on Reuse Distance and Cache Misses 100
3.3.1 Cache misses versus Reuse Distance 100
3.3.2 Effect of Compiler Optimization on Reuse Distance Distribution 102
3.4 Related Work 104
3.5 Summary 106

4 The Reuse Distance Equations 107
4.1 Polyhedral Model 108
4.1.1 Polytopes and Polyhedra 108
4.1.2 Parameterized Polytopes 113
4.1.3 Presburger Arithmetic 117
4.1.4 Program Representation in the Polyhedral Model 118
4.2 Reuse Distance Equations 122
4.2.1 Reuse pair 122
4.2.2 Accessed Data Set of a Reuse Pair 123
4.2.3 Reuse Distance of a Reuse Pair 125
4.2.4 Example: Cholesky Factorization 127
4.2.5 Extensions to Cache Equations 127
4.3 Enumerating Parameterized Polytopes 132
4.3.1 Ehrhart's Theory 132

4.3.2 Interpolation Method 136
4.3.3 Limitations of Interpolation Method 137
4.4 Enumerating Polytopes using Barvinok's Decomposition 143
4.4.1 Non-Parameterized Polytope Counting 143
4.4.2 Parameterized Polytope Counting 149
4.5 Enumerating Parametric Presburger Formula 153
4.5.1 Conversion to Disjoint Disjunctive Normal Form 155
4.5.2 Enumerating Sets of Disjoint Polytopes 156
4.5.3 Simplified Enumerations 161
4.6 Experiments 162
4.6.1 Reuse Distance Calculation 163
4.6.2 Taking into account Cache Line Size 169
4.7 Related Work 170
4.8 Summary 173

5 Cache Hint Selection 175
5.1 Cache Hints in EPIC Architectures 175
5.1.1 Cache Hints in the HPL-PD Architecture 176
5.1.2 Cache Hints in the IA-64 Architecture 177
5.1.3 Static and Dynamic Hints 179
5.2 Static Hint Selection 179
5.2.1 Cache Hint Selection for a Memory Access 179
5.2.2 Cache Hint Selection for a Memory Instruction . . 180
5.3 Dynamic Hint Selection 182
5.3.1 Code Generation 185
5.3.2 Overhead Reduction 186
5.4 Experiments 187
5.4.1 Static Cache Hint Implementation 187
5.4.2 Static Hints Experiments 191
5.4.3 Dynamic Hint Experiments 194
5.4.4 Discussion 199
5.5 Related Work 201
5.6 Summary 202

6 Cache Bottleneck Visualization 205
6.1 Access Stream Visualization 206
6.2 Low-Locality Reuse Visualization 211
6.2.1 Platform-Independent Cache Optimizations . . . 211
6.2.2 Locality Metrics 212
6.2.3 Implementation 213

6.3 Experiments 215
6.3.1 Mcf 215
6.3.2 Art 216
6.3.3 Equake 218
6.3.4 Discussion 220
6.4 Related Work 221
6.5 Summary 223

7 Conclusion **225**
7.1 Summary and Contributions 225
7.2 Future Research Directions 228

A Computed Forward Reuse Distances: Examples **231**
A.1 Cholesky 231
A.2 Matrix Multiplication 234
A.3 Gauss-Jordan 237

List of Tables

I Dutch Summary: Softwaretechnieken ter verbetering van datalokaliteit en cachegedrag **1**

5.1 Cache missers voor de LRU- en LRU+CH-vervangingsalgoritmen. 43
5.2 Overhead van dynamische doelhints 44
5.3 Percentage verkeerd voorspelde voorwaartse hergebruiksafstanden en doelhints door enkel de dominante Ehrhart-polynomen te beschouwen. 44

II Software Methods to Improve Data Locality and Cache Behavior **59**

4.1 The parametric vertices of the polyhedron in figure 4.4. . 117
4.2 The validity domains of the parametric polyhedron in figure 4.4. 117
4.3 The validity domains of the parametric polytope in equation (4.42). 139
4.4 Number of references and maximum loop depth of the benchmark programs. 164
4.5 Total number of iteration domains before and after simplification. 166
4.6 Number of polytopes enumerated for FRD calculation . . 167
4.7 Number of polytopes enumerated for BRD calculation . 167
4.8 Number of constant, linear and quadratic dominant polynomials 168

4.9 The number of Presburger formulas describing value-based dependences that cannot be processed by the Omega library. 169
4.10 Polytope counting for cache line size=4. 170
4.11 Comparison between different analytical cache equation models 171

5.1 The effect of target hints in the Itanium-2 processor . . . 178
5.2 Expected cost of a wrong cache hint. 182
5.3 Speedup after source hints, both with and without software pipelining 193
5.4 Cache miss rates for LRU replacement and LRU+CH replacement policies. 196
5.5 Overhead of dynamic forward reuse distance computation 197
5.6 Fraction of incorrect forward reuse distances in dominant domains 198
5.7 Comparison of profile-based static hints versus analysis-based dynamic hints. 199

6.1 Cache sizes and associativity for the different processors. 221

List of Figures

I Dutch Summary: Softwaretechnieken ter verbetering van datalokaliteit en cachegedrag **1**

1.1 Oorzaken van prestatieverlies op een Itanium1 multiprocessor voor de SPEC2000 benchmark. 4
1.2 Het percentage van de datacache-missers dat veroorzaakt wordt door beperkte cachecapaciteit, voor de SPEC2000-programma's 5

2.1 Getegelde en niet-getegelde versie van de matrixvermenigvuldiging 10
2.2 Illustratie van cacheorganizatie 11
2.3 De remap-draad brengt de volgende tegelverzameling in de cache, terwijl de rekendraad de huidige tegelverzameling verwekt. 12
2.4 Gepijplijnde rekenwijze bij cache remapping. 12
2.5 Eén van de Q functies die een enkel element verplaatsen 13
2.6 De remap functie. De `nta` cache hint verhindert allocatie in de cache. 13
2.7 Lustransformatie die de rekendraad en remap-draad tot een enkele draad weeft. 14
2.8 Vergelijking van de prestatie van verschillende tegel-algoritmen 16

3.1 Miss rate in functie van hergebruiksafstand voor SPEC95FP. 19
3.2 Het aantal cachemissers voor en na optimalisatie, in functie van de hergebruiksafstand 21

4.1 Grafische voorstelling van de polytoop die bepaald wordt door $2x + y \geq 1 \wedge x - 3y \geq 3 \wedge x - y \leq 7$ 24
4.2 De grafische voorstelling van een geparameteriseerde polytoop 25
4.3 Voorbeeld van een programma in het polyhedrale model. 26
4.4 Voorbeeldje van berekende hergebruiksparen 28
4.5 Voorbeeld van berekening van aangesproken data tussen gebruik en hergebruik 30
4.6 Voorbeeld van hergebruiksafstandsberekening 31

5.1 Effect van de cache hints in de load-instructie `LD_C2_C3` . 36
5.2 Voorbeeld van een cumulatieve hergebruiksafstandsdistributie 37
5.3 Voorbeeld van een hergebruiksafstandsdistributie waarvoor een goede statisch cache hint selectie niet mogelijk is. 38
5.4 Het programma in figuur 5.3 met dynamische doelhintberekening voor referentie A(i). 40
5.5 Versnelling met statische cache hints 42

6.1 Visualisatie van de belangrijkste lange-afstandshergebruiken in mcf. 50
6.2 Versnelling op verschillende architecturen. 51
6.3 Hergebruiksafstanden voor en na optimalisatie. 52

II Software Methods to Improve Data Locality and Cache Behavior 59

1.1 Causes of performance loss on an Itanium1 multiprocessor for the SPEC2000 benchmark 63
1.2 The percentage of data cache misses caused by limited capacity for varying cache size and associativity 64

2.1 A tiled loop nest 74
2.2 Non-tiled and tiled version of the matrix multiplication. 75
2.3 The histogram of reference distances for the matrix multiplication ($N = 256$) before and after tiling (tile size=20). 76
2.4 Illustration of cache organization 76

2.5 Example of a tile set in the matrix multiplication that generates conflict misses 77
2.6 The remap thread puts the next tile set in the cache while the original thread processes the current tile set 78
2.7 The pipelined nature of cache remapping 79
2.8 Overview of the interaction between the cache shadow, the process thread and the remap thread. 80
2.9 One of the Q functions that remap one element 82
2.10 The remap function 83
2.11 Program transformation to efficiently interweave and schedule both processing and remapping threads into one . . . 83
2.12 Smoothed plot of the performance of several tiled matrix multiplications for dimension 20 to 400 84
2.13 Comparison of performance of different tiling algorithms, for matrix dimensions 200 to 400 85

3.1 Cache model 90
3.2 Example memory access stream and corresponding reuse distances 91
3.3 Example of backward and forward reuse distance distributions of reference r_3 in the example in figure 3.2. 93
3.4 Treap representation of LRU stack after processing trace IAKJCEDFAAGHHFBFK 96
3.5 Linked list representation of LRU stack after processing trace IAKJCEDFAAGHHFBFK 98
3.6 Reuse distance calculation speed for the treap-based and the log2border method 99
3.7 Miss rate versus reuse distance for SPEC95FP. 101
3.8 The number of cache misses before and after optimization, in function of their reuse distance 103
3.9 Reuse distance calculation speed versus reference distance calculation speed 105

4.1 Geometrical representation of the polyhedron defined by $2x + y \geq 1 \wedge x - 3y \geq 3$ 109
4.2 Geometrical representation of the integer polytope defined by $2x + y \geq 1 \wedge x - 3y \geq 3 \wedge x - y \leq 7$ 110
4.3 Geometrical representation of the integer polytope defined by $2x + y \geq 1 \geq 2x + y \wedge -1 \leq y \leq 3$ 112
4.4 The geometrical representation of a parameterized polytope 114

4.5 Example of combined data-parameter space, and its faces. 116
4.6 Example program. 120
4.7 Geometric representation of division constraint $\alpha = \lfloor \frac{x}{4} \rfloor$. 121
4.8 Geometric representation of modulo constraints $4\alpha \leq x \leq 4\alpha + 3 \wedge \beta = x - 4\alpha$ 121
4.9 Example of reuse pair calculation 124
4.10 Example of accessed data set calculation 126
4.11 Example of reuse distance calculation 128
4.12 Cholesky factorization 129
4.13 Example of wildly varying reuse distances generated by single reference 130
4.14 Example of varying reuse distance for different data sizes 131
4.15 Example of a homothetic polyhedron 133
4.16 Geometrical representation of the validity domains of equation (4.42). 140
4.17 Matrix multiplication 141
4.18 Intermediate accesses between reuses of `A(i,k)` at iterations points (i, j, k) and $(i, j + 1, k)$. 141
4.19 Example polytope for which the generating function is $1 + x_1 + x_1^2 + x_2 + x_1x_2 + x_2^2$. 144
4.20 Barvinok's decomposition of $\mathrm{cone}(P, \mathbf{v_1})$ into two unimodular cones K_1 and K_2 (indicated by two different shades of grey). 147
4.21 Geometrical representation of the overlapping iteration domains 4.97, 4.98, 4.99, 4.100, 4.101. 159
4.22 Code for which the calculation of $\mathsf{BRD}(\mathtt{A(k)})$ illustrates the simplification of enumerations. 162
4.23 Relative computation time of the different steps in BRD calculation 165

5.1 Example of the effect of cache hints in the load instruction `LD_C2_C3` 176
5.2 The semantics of target cache hints in the IA-64 architecture 178
5.3 Example cumulative reuse distance distribution for an instruction 181
5.4 Example reuse distance distribution for which good static cache hint selection is impossible 183
5.5 The program in figure 5.4 with dynamic hint calculation for reference A(i). 184

5.6 Source code from MGRID, and the associated backward reuse distance distribution for reference `R(I1,I2,I3)`. . 188
5.7 Software pipelined schedule without source hints 189
5.8 Software pipelined schedule with source hints 190
5.9 Speedup after static source and target hint generation . . 192
5.10 Relative execution time and code size for static hints, plotted according to original code size. 193
5.11 Scalability of code size and execution time overhead for dynamic target hints based on the dominant domains. . . 198

6.1 Matrix multiplication trace visualization 207
6.2 FFT trace visualization 208
6.3 Highlighted Tomcatv source code 208
6.4 Tomcatv trace visualization 209
6.5 VCG-generated visualization of major long distance reuses in mcf. 215
6.6 A zoom in on the major long reuse distance in 181.mcf . 216
6.7 A zoom-out view of the major long reuse distances in 179.art 217
6.8 A zoom-out view of the major long reuse distances in 183.equake. 219
6.9 Speedups on different architectures. 220
6.10 Reuse distance distributions before and after optimization. 222

A.1 Cholesky factorization 231
A.2 matrix multiplication 234
A.3 Gauss-Jordan elimination 237

List of Notations

L	A loop nest L	73
$\mathsf{Td}(\mathcal{L})$	The tiled loops of a tiled loop nest $\mathcal{L}$	73
$\mathsf{Ti}(\mathcal{L})$	The tiling loops of a tiled loop nest $\mathcal{L}$	73
$\mathsf{Ml}(\mathcal{L}, N)$	The memory lines accessed by loop nest $\mathcal{L}$ that map to cache set N	75
$\mathsf{C_s}$	Cache size	90
$\mathsf{L_s}$	Line size	90
A	Cache associativity	90
$\mathsf{N_s}$	Number of cache sets	90
$a, b, c, \ldots$	Memory accesses a, b, c, ...	91
$r, s, t, \ldots$	Memory references r, s, t, ...	91
$\langle a, b \rangle_L$	Reuse pair with memory line size L	92
$\mathsf{ADS}\langle a, b \rangle_L$	Accessed data set of $\langle a, b \rangle_L$	92
$\mathsf{RD}(\langle a, b \rangle_L)$	Reuse distance of $\langle a, b \rangle_L$	92
$\mathsf{BRD}_L(b)$	Backward reuse distance of b	92
$\mathsf{FRD}_L(a)$	Forward reuse distance of a	92
$\mathsf{RDD}_L(r, s)$	Reuse distance distribution of reference pair r, s	93
$\mathsf{BRDD}_L(r)$	Backward reuse distance distribution of reference r	93
$\mathsf{FRDD}_L(r)$	Forward reuse distance distribution of reference r	93
P	Polyhedron or polytope P.	108
C	Polyhedral cone C.	110
$\mathcal{R}$	The set of all references in a program	119
$\mathcal{V}$	The set of all variables in a program	119
$\mathsf{IS}(r)$	The iteration space of reference r	119
$r@i$	The memory location accessed by reference r at iteration i	119

$i_r \lessdot j_s$	Iteration i of reference r executes before iteration j of reference s	119
$\mathcal{P}$	The set of all program parameters	119
$\mathcal{E}(P;\mathbf{N})$	The enumerator of polytope P with parameters $\mathbf{N}$	135
$\mathcal{B}(K)$	The unimodular decomposition of cone K	145

List of Definitions

tiled loops 73
tiling loops 73
iteration tile 73
data tile 73
tile set 73
cache size 75
line size 75
associativity 75
cache sets 75
memory line 75
memory reference 91
memory access 91
memory line 91
reuse pair 92
accessed data set 92
reuse distance 92
backward reuse distance 92
forward reuse distance 92
reuse distance distribution 93
backward reuse distance distribution 93
forward reuse distance distribution 93
LRU replacement policy 94
cold miss 95
conflict miss 95
capacity miss 95
convex polyhedron 108
integer polyhedron 108
line 109
ray 109
vertex 109

polytope 110
polyhedral cone 110
generator 110
unimodular generator 110
unimodular cone 110
supporting cone 110
implicit equality 111
affine hull 111
dimension of a polyhedron 111
face of a polyhedron 112
face-lattice 112
denominator of a vertex 113
denominator of a polyhedron 113
parameterized polyhedron 113
validity domain 117
Presburger formula 118
homothetic polyhedron 133
periodic number 134
q-periodic number 134
pseudo-polynomial 135
pseudo-period 135
Ehrhart polynomial 135
degenerate domain 138
dominant domain 168
dominant polynomial 168
memory line utilization 213

Deel I

Dutch Summary: Softwaretechnieken ter verbetering van datalokaliteit en cachegedrag

Hoofdstuk 1

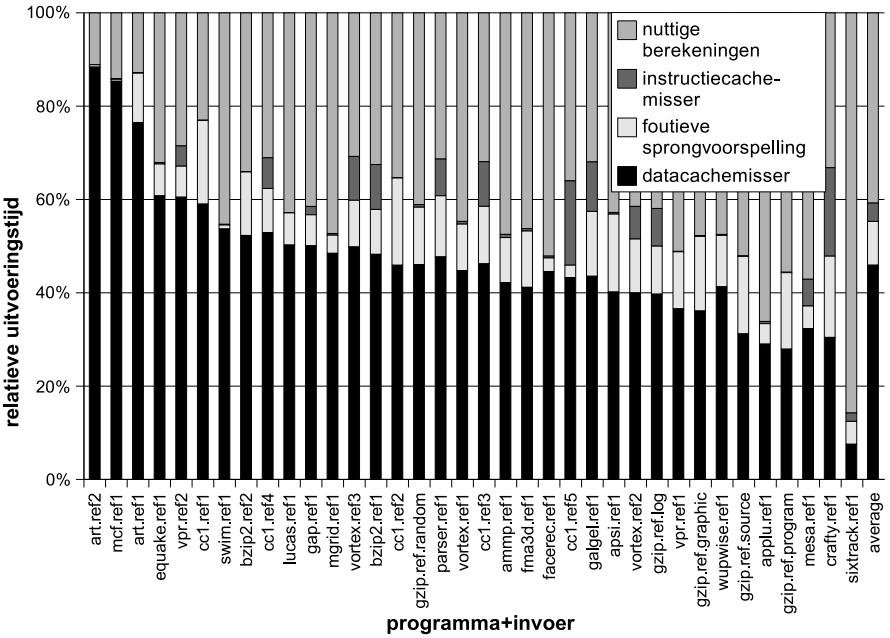
Inleiding

Iedere computer bestaat uit minstens twee onderdelen: een processor en geheugen. Elke programmeur wenst dat zijn computer snel rekent, en veel geheugen bevat, zodat hij zo veel mogelijk problemen kan verwerken. Snelle geheugens zijn echter duur, en grote geheugens kunnen nooit zo snel gemaakt worden als kleine geheugens. Daarom bevatten bijna alle computers een geheugenhiërarchie. Om efficient te rekenen en de data snel genoeg aan de processor aan te bieden, moet de meerderheid van alle geheugentoegangen in de kleinste niveau's van de geheugenhiërarchie gevonden worden.

Processors worden ieder jaar ongeveer 55% sneller. Daartegenover worden RAM-geheugens slechts ongeveer 7% per jaar sneller [83]. Hierdoor groeit de snelheidskloof tussen processor en geheugen tegen een tempo van 45% per jaar. Om die kloof te overbruggen moeten er steeds krachtiger optimalisaties gebruikt worden om zoveel mogelijk data in de snelste geheugenniveau's te houden.

1.1 Achtergrond

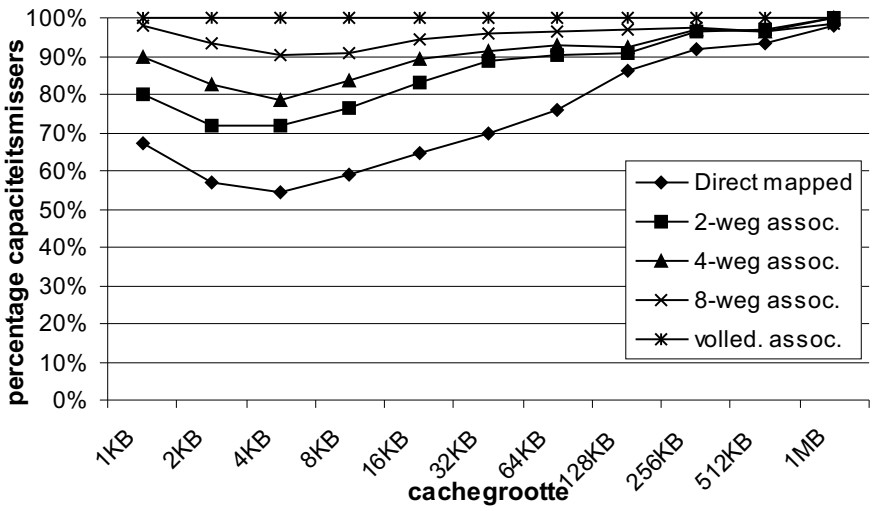
Reeds in de eerste elektronische computers waren trage hoofdgeheugens een belangrijke vertragende factor. Begin de jaren '50 werden dan ook geheugenhiërarchieën geïntroduceerd. Bij deze vroege computers moesten de programma's instructies bevatten die expliciet de data verplaatst tussen de verschillende lagen in de hiërarchie. In de meeste moderne computers bestaan de meeste lagen in de hiërarchie uit caches, die op automatische wijze de data op een gepast niveau in de hiërarchie plaatsen. De caches bevatten hardware die probeert te



Figuur 1.1: Oorzaken van prestatieverlies op een Itanium1 multiprocessor voor de SPEC2000 benchmark.

voorspellen welke data in de nabije toekomst het meest gebruikt wordt, en op basis daarvan wordt beslist of de data al dan niet in een bepaald cacheniveau gehouden wordt.

Aangezien het efficiënt gebruik van de caches zo belangrijk is voor het snel werken van computers, is er reeds veel onderzoek naar gebeurd. De voorgestelde verbeteringen gaan van hardwareverbeteringen, over micro-architecturale aanpassingen, compileroptimalisaties en besturingssysteemverbeteringen tot optimalisaties op het algoritmische niveau. Ondanks het werk van vele onderzoekers blijven cachemissers een van de belangrijkste oorzaken van traag werkende computers. Als illustratie wordt in figuur 1.1 voor de SPEC2000 programma's getoond welke percentage van hun uitvoeringstijd verloren gaat aan verschillende oorzaken. De Itanium-processor heeft 3 cacheniveau's, waaraan 85% van alle transistors op de chip worden opgeofferd. Verder werd Intels state-of-the-art optimaliserende compiler gebruikt om de programma's te compileren. Ondanks de doorgedreven optimalisaties zorgen de cachemissers ervoor dat de processor bijna de helft van de tijd staat te wachten en geen nuttig werk uitvoert.



Figuur 1.2: Het percentage van de data cache-missers dat veroorzaakt wordt door beperkte cachecapaciteit, voor de SPEC2000-programma's, zoals opgemeten door Cantin en Hill [37].

Caches zijn effectief omdat data in typische programma's vele malen gebruikt worden tijdens een enkele uitvoering. Bovendien worden meestal gegevens hergebruikt die recent voor het laatst gebruikt werden. Deze eigenschap wordt de *lokaliteit* van het programma genoemd. Wanneer de lokaliteit niet kan uitgebuit worden door caches treden er missers op. Iedere misser kan geclassificeerd worden in een van de volgende drie klassen. De misser is een *koude misser* indien de data voor het eerst werd aangesproken. Het is een *capaciteitsmisser* wanneer er zoveel data werd aangesproken tussen gebruik en hergebruik, dat die niet allemaal tegelijk in de cache kan passen. De *conflictmissers* zijn de missers waarbij gebruik en hergebruik elkaar snel opvolgen, en de misser een gevolg is van het beperkt aantal lokaties waarin een data-element kan opgeslagen worden in set-associatieve of direct mapped caches. In typische programma's is het aantal koude missers zeer klein. Figuur 1.2 toont dat voor de SPEC2000 programma's en voor caches die typisch in huidige processors gebruikt worden (2-weg-associatief of meer, groter dan 8KB), minstens 80% van alle missers capaciteitsmissers zijn. Op hardware-niveau kunnen capaciteitsmissers enkel weggewerkt worden door grotere, en dus ook tragere, caches

te gebruiken [83]. Het is dus aangewezen om de overheersende capaciteitsmissers te bestrijden door software-optimalisaties door te voeren.

1.2 Overzicht

Deze verhandeling behandelt softwaretechnieken om het cachegedrag van datatoegangen te verbeteren. De nadruk ligt op het wegwerken van capaciteitsmissers aangezien die de meest voorkomende soort zijn.

In hoofdstuk 2 wordt *cache remapping* voorgesteld. Cache remapping is een techniek die verschillende bestaande cache-optimalisaties combineert en tracht ervoor te zorgen dat de processor nooit hoeft te wachten op data uit trage geheugenniveau's.

In hoofdstuk 3 wordt de *hergebruiksafstand* geïntroduceerd als maat voor datalokaliteit. De hergebruiksafstand voorspelt het cachegedrag van geheugentoegangen perfect voor volledig-associatieve caches. Voor minder-associatieve caches wordt het cachegedrag ook behoorlijk accuraat voorspeld. Verder wordt er nagegaan in welke mate cachemissers door automatische programma-optimalisaties in een state-of-the-art compiler kunnen verwijderd worden. Daaruit blijkt dat de compiler 30% van de conflictmissers verwijdert, maar slechts 1% van de capaciteitsmissers. Automatische compiler-optimalisaties blijken niet in staat te zijn om missers met lange hergebruiksafstand te verwijderen.

In de daaropvolgende hoofdstukken wordt de aandacht gericht op het bepalen en verwijderen van capaciteitsmissers. Capaciteitsmissers worden speciaal geviseerd omdat: (a) zij het dominante type misser zijn; (b) capaciteitsmissers enkel door softwaretechnieken weggewerkt kunnen worden, en conflictmissers ook door micro-architecturale optimalisaties kunnen verwijderd worden; (c) de huidige compileroptimalisaties wel in staat zijn een aanzienlijk deel van de conflictmissers te verwijderen, maar nauwelijks capaciteitsmissers kunnen wegwerken. Kortom, capaciteitsmissers lijken inherent moeilijker te verwijderen, terwijl ze wel het meest voorkomende type misser zijn.

Naast het profileren van hergebruiksafstanden, zoals beschreven in hoofdstuk 3, kunnen hergebruiksafstanden ook berekend worden voor programma's in het polyhedrale model. In hoofdstuk 4 worden de *hergebruiksafstandsvergelijkingen* voorgesteld, die de basis vormen voor

de berekening van de hergebruiksafstanden.

In hoofdstuk 5 worden cachehints geselecteerd op basis van de hergebruiksafstand. Cachehints zijn annotaties op geheugeninstructies zoals loads, stores en prefetches. Er zijn twee soorten cachehints: *bron-* en *doel*-hints. De bronhints duiden het snelste cacheniveau aan waar de opgevraagde data verwacht wordt aanwezig te zijn. De doelhints duiden het snelste cacheniveau aan waarvan verwacht wordt dat de data er zal blijven zitten tot de volgende keer dat het aangesproken wordt. De bronhints worden door de instructiescheduler gebruikt om de latentie van load-instructies beter te kunnen schatten. Zonder bronhints veronderstelt een instructiescheduler dat iedere geheugeninstructie aanleiding geven tot L1 cachetreffers. Met behulp van de bronhints krijgt de scheduler een beter zicht op de werkelijke latentie van de instructies, en worden de instructies beter geplaatst. De doelhints kunnen het vervangingsalgoritme in de cache verbeteren door data enkel te bewaren in de niveaus waarvan verwacht wordt dat ze de data ook tot het volgend gebruik zullen bijhouden. Op die manier wordt cachevervuiling verminderd.

Er worden twee alternatieve methodes voorgesteld om cachehints te genereren. De eerste methode is gebaseerd op geprofileerde hergebruiksafstandsdistributies en genereren *statische hints*, i.e. er wordt een vaste hint geselecteerd voor iedere instructie. De tweede methode is gebaseerd op de hergebruiksafstandsvergelijkingen en genereert *dynamische hints*, i.e. voor iedere uitvoering van een geheugeninstructie wordt de meest geschikte hint berekend. De dynamische hints vereisen een beperkt aantal extra berekeningen at run-time. Deze overhead wordt weggewerkt door de rekening te houden met de structuur van de oplossingen van de hergebruiksafstandvergelijkingen. De cache hint-generatie werd geïmplementeerd in de ORC-compiler voor de Itanium-processor, en werd geëvalueerd aan de hand van een aantal numerieke programma's en een aantal programma's met vooral pointers en dynamische datastructuren.

In hoofdstuk 6 wordt aanvaard dat compileroptimalisaties beperkt zijn in het aantal capaciteitsmissers die ze kunnen wegwerken. Die capaciteitsmissers gebeuren immers op grote hergebruiksafstand, en een globaal overzicht over de programma-uitvoering is nodig om zowel het gebruik als het hergebruik te kunnen beschouwen. In dit hoofdstuk wordt het verwijderen van capaciteitsmissers overgelaten aan de programmeur, die een beter zicht heeft op het globaal programmage-

drag. Bovendien kan een programmeur vaak het programma veel verregaander veranderen dan een compiler. Het cachegedrag is echter niet duidelijk te zien in de broncode. Daarom wordt er een visualisatie voorgesteld van de lange-afstandshergebruiken. In vergelijkingen met vroegere hulpmiddelen die cachemissers tonen aan de programmeur heeft deze visualisatie het voordeel dat de programmeur gestuurd wordt in de richting van platform-onafhankelijke optimalisaties. Als test werd de visualisatie toegepast op drie programma's uit de SPEC2000 benchmark. Nadat er een aantal relatief kleine aanpassingen in de broncode werden doorgevoerd, gebaseerd op de visualisatie, liepen de programma's gemiddeld 3 maal sneller op verschillende computerplatformen, gaande van PC's tot servers, die Athlon, Alpha en Itanium processors bevatten.

In hoofdstuk 7 worden de belangrijkste conclusies en bijdragen samengevat.

Hoofdstuk 2

Cache Remapping

Cache remapping tracht ervoor te zorgen dat de processor nooit moet wachten op gegevens uit trage geheugenniveaus. Om dit te bereiken worden verschillende optimalisaties gecombineerd. Vooreerst wordt het aantal capaciteitsmissers verminderd door lustegeling. Daarna worden de conflictmissers aangepakt door de data layout at run-time aan te passen. De overblijvende missers worden verborgen door middel van prefetching. Het herlayouten van data at run time laat toe om onrechtstreeks te bepalen welke data in de cache bewaard blijft, en welke niet.

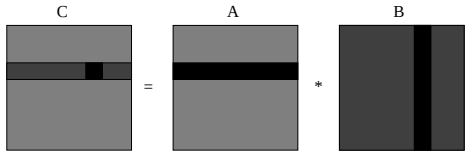
Het prefetchen en herlayouten gebeurt in een aparte remap-draad*, terwijl de berekeningen in de* rekendraad *worden gedaan. Voor lussen die met een vaste tegelgrootte werden getegeld, kunnen deze draden tijdens de compilatie door elkaar geweven worden tot een enkele uitvoeringsdraad, zodat het programma, naast op meerdradige processors, ook op enkeldradige processors kan uitgevoerd worden.*

In vergelijking met andere methodes om het aantal cachemissers na tegeling te verminderen, leidt cache remapping tot een snellere uitvoering door de combinatie van herlayouten en prefetching.

2.1 Lustegeling

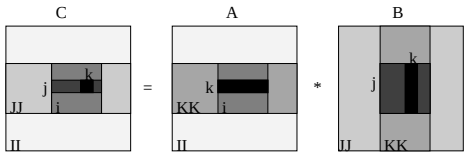
Definitie 1. *Lustegeling transformeert een n-diep lusnest in een $2n$-diep lusnest. De* **getegelde** *lussen in het $2n$-diep lusnest zijn de n binnenste lussen. De n buitenste lussen zijn de* **tegelende** *lussen. Een* **iteratietegel** *bestaat uit de iteraties die gedurende een uitvoering van de getegelde lussen worden uitgevoerd. Een* **datategel** *is het deel van een array dat door een it-*

```
DO i=1,N
 DO j=1,N
  DO k=1,N
   C(i,j) += A(i,k) * B(k,j)
```

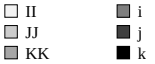


(a) Matrixvermenigvuldiging

```
DO II=1,N,B1
 DO JJ=1,N,B2
  DO KK=1,N,B3
   DO i=1,min(N,II)
    DO j=1,min(N,JJ)
     DO k=1,min(N,KK)
      C(i,j) += A(i,k) * B(k,j)
```



(b) Getegelde matrixvermenigvuldiging



Figuur 2.1: Getegelde en niet-getegelde versie van de matrixvermenigvuldiging. De verschillende grijstinten tonen welke delen van de arrays door welke lussen aangesproken worden. In de getegelde code worden er meer berekening per hoeveelheid data gedaan.

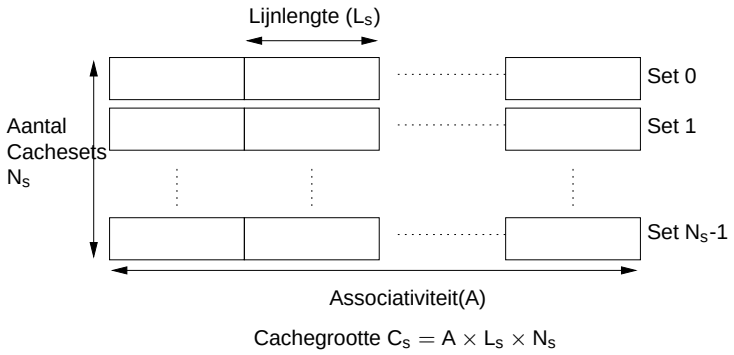
eratietegel wordt aangesproken. Een **tegelverzameling** *is de unie van de datategels van alle arrays.*

Een voorbeeld van een getegelde matrixvermenigvuldiging wordt getoond in figuur 2.1.

2.2 Cachegeheugen

Een cache wordt gekarakteriseerd door middel van een quadruppel (C_s, N_s, A, L_s):

Definitie 2. *De* **cachegrootte** C_s *duidt het aantal bytes in de cache aan. De* **lijnlengte** *duidt het aantal opeenvolgende bytes aan dat uit het hoofdgeheugen wordt gehaald bij iedere misser. Een* **geheugenlijn** *is een aangesloten blok bytes in het geheugen dat in één beweging in de cache gebracht wordt tijdens een misser.* N_s *duidt het aantal cache sets aan. De* **associativiteit** A *bepaalt het aantal cachelijnen in een cacheset. Deze parameters zijn gerelateerd via de formule* $C_s = N_s \times A \times L_s$.



Figuur 2.2: Illustratie van cacheorganizatie

De cacheorganisatie wordt geïllustreerd in figuur 2.2.

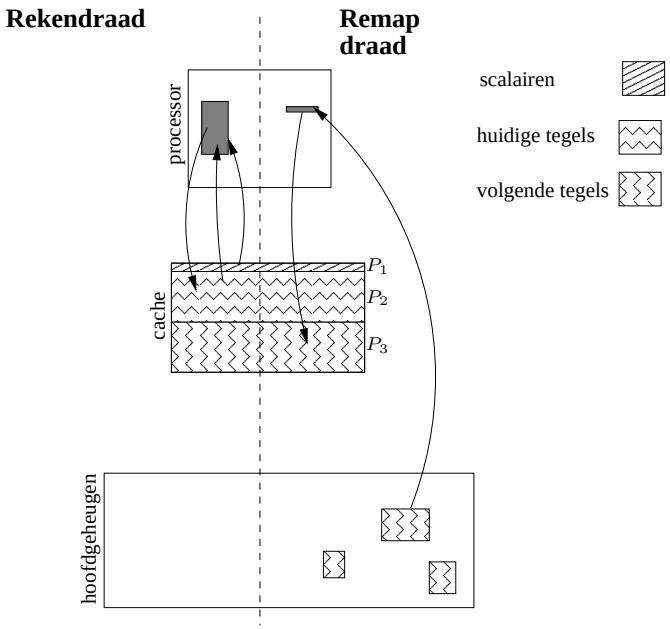
Beschouw een getegeld lusnest en een cacheset N. Wanneer het aantal geheugenlijnen in een tegelverzameling dat mapt naar de cacheset N groter is dan A, treden er conflictmissers op. Bij cache remapping wordt de tegelverzameling gekopieerd in een aangesloten blok van C_s bytes groot. Op die manier worden er juist A geheugenlijnen afgebeeld op iedere cacheset, en treden er geen conflictmissers op.

2.3 Cache Remapping vanuit Vogelperspectief

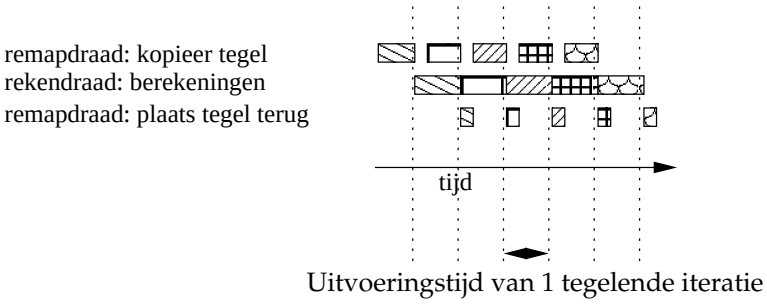
Cache remapping voegt een remap-draad toe aan het programma. De remap-draad kopieert de volgende tegelverzameling in opeenvolgende geheugenlokaties, terwijl de rekendraad de huidige tegelverzameling verwerkt (zie figuur 2.3). Beschouw een iteratiepunt i van de tegelende lussen. De twee draden werken op gepijplijnde wijze (zie figuur 2.4):

- De *rekendraad* verwerkt tegel i.
- De *remap-draad* kopieer tegelverzameling $i + 1$ naar de cache. Indien er data is die gewijzigd werd in tegelverzameling $i-1$, wordt die eerst teruggekopieerd naar zijn oorspronkelijke lokatie in het hoofdgeheugen.

Beide draden synchroniseren tussen twee iteraties van de tegelende lussen. De tegelgroottes worden zodanig gekozen dat twee opeenvolgende tegelverzamelings gelijktijdig in de cache kunnen geplaatst worden, samen met eventuele scalaire data die in alle iteratietegels wordt



Figuur 2.3: De remap-draad brengt de volgende tegelverzameling in de cache, terwijl de rekendraad de huidige tegelverzameling verwekt. In de volgende stap zullen de rekendraad en de remap-draad respectievelijk P_3 en P_2 aanspreken.



Figuur 2.4: Gepijplijnde rekenwijze bij cache remapping.

```
remapA(int iter,A,p) {
  i1 = de_coalesce(iter);
  i2 = de_coalesce(iter);
  remap(p+i1*B2+i2, A[i1+II,i2+JJ]);
}
```

Figuur 2.5: Eén van de Q functies die een enkel element verplaatsen

```
remap(double* x, double* y)
{
   ldfd.nta  r1,y
   stfd      x,r1
}
```

Figuur 2.6: De remap functie. De `nta` cache hint verhindert allocatie in de cache.

gebruikt. De scalaire data wordt in cacheparitie P_1 geplaatst. Partities P_2 en P_3 zijn even groot en bevatten elk een tegelverzameling.

2.4 Cache Remapping vanuit Kikvorsperspectief

2.4.1 Softwarecontrole over het Cachegedrag

Tijdens de opstartfase wordt een blok van C_s bytes gereserveerd, dat de **cacheschaduw** wordt genoemd. De gebieden P_1, P_2 en P_3 worden in de cacheschaduw geplaatst. Er bestaat een bijectie tussen de lokaties in de cacheschaduw en de cache. De inhoud van de cacheschaduw zal altijd in de cache gehouden worden. Dit wordt gegarandeerd door cache hints te gebruiken (zie hoofdstuk 5), zodat voor alle geheugentoegangen naar gebieden buiten de cacheschaduw, de data niet in de cache gebracht worden.

2.4.2 Draadscheduling op een Enkeldradige Processor

Om de overhead van het herlayouten en verplaatsen van tegels te verkleinen dienen de rekendraad en de remap-draad gelijktijdig uit te voeren. Op een superscalaire enkeldradige processor kan dit gedaan worden door instructies uit beide draden door elkaar te weven. Dit gebeurt door volgende lustransformaties.

```
swap(p2,p3)
iter=0
do i = II,II+Q_1^{B_1}-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      H(i,j,k,p2) /* lichaam r */
      ...  /* maal ontrold */
      H(i,j,k+r-1,p2)
          /* remap code */
      remapA(iter++,A,p3)
iter=0
do i = II+Q_1^{B_1},II+Q_1^{B_1}+Q_2^{B_1}-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      ...
      remapB(iter++,B,p3)
...
```

Figuur 2.7: Lustransformatie die de rekendraad en remap-draad tot een enkele draad weeft. `p2` en `p3` zijn de startadressen van P_2 en P_3. Er wordt verondersteld dat na inlining de instructiescheduler genoeg onafhankelijke instructies tussen beide instructies in `remap` plaatst om de latentie van de hoofdgeheugentoegang te verbergen met parallelle instructies.

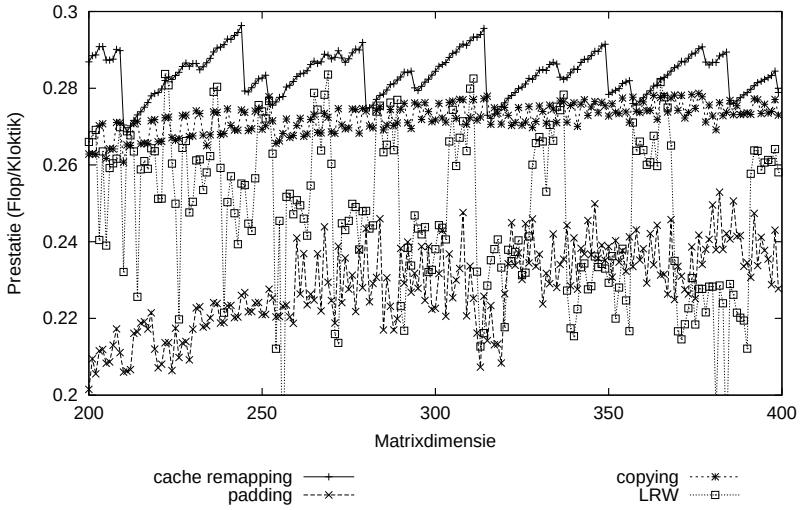
De remap-draad bestaat oorspronkelijk uit Q lusnesten, waarbij ieder lusnest een enkele datategel verplaatst tussen het hoofdgeheugen en de cacheschaduw. Q_i^r is het aantal array-elementen dat door lusnest i verplaatst wordt. In een eerste stap worden deze lusnesten gecoalesced [137], zodat ze omgevormd worden tot enkelvoudige lussen. Daarna wordt het luslichaam geplaatst in een ingelijnde remap-functie (zie bijvoorbeeld `remapA` in figuur 2.5). De binneste lus van het getegelde lusnest in de rekendraad word $\lfloor r \rfloor$ maal ontrold, waarbij r het aantal iteraties van de getegelde lussen is gedeeld door het aantal elementen in een tegelverzameling. Daarna wordt een oproep naar een geschikte remap-functie ingevoegd. Tijdens het compileren, is geweten hoeveel maal iedere remap-functie moet uitgevoerd worden. De buitenste lus van de getegelde lussen wordt opgesplitst in Q delen. In ieder deel wordt een andere remap-functie opgeroepen. Het aantal iteraties van ieder deel wordt zo gekozen dat iedere remap-functie minstens Q_i^r maal wordt opgeroepen: $Q_i^{B_1} * B_2 * \left\lfloor \frac{B_3}{r} \right\rfloor \geq Q_i^r$.

2.5 Implementatie

Cache remapping werd toegepast op een matrix-vermenigvuldiging, en vergeleken met een aantal andere methodes uit de recente literatuur om het aantal cachemissers in getegelde lusnesten te verminderen, zoals padding [134], copying [171] en tegelgrootte-selectie [106]. De vergelijking werd gemaakt door het programma te compileren en uit te voeren in de Trimaran [173]-omgeving. In figuur 2.8 wordt de prestaties van de verschillende algoritmen getoond voor matrixgroottes tussen 200×200 en 400×400. Daaruit blijkt dat cache remapping aanleiding geeft tot een prestatie die gemiddeld 5% beter is dan de beste alternatieve methode.

2.6 Samenvatting

In dit hoofdstuk werd cache remapping voorgesteld. Deze programmaoptimalisatie combineert verschillende methodes om het cachegedrag te verbeteren, zoals lustegeling, aanpassen van data layout at run-time, een cacheschaduw en cache hints om de inhoud van de cache vanuit de software te controleren, en prefetching. In vergelijking met de aanverwante technieken die in de literatuur werden voorgesteld, levert cache



Figuur 2.8: De prestatie van cache remapping, padding [134], copying [171] and tegelgrootteselectie (LRW) [106] voor matrixdimensies van 200 tot 400. Cache remapping is steeds minstens even snel als de andere algoritmen. In het beste geval is het 10% sneller.

remapping een extra gemiddelde prestatiewinst van 5% op.

Hoofdstuk 3

De Hergebruiksafstand als Maat voor Datalokaliteit

Vooraleer het aantal capaciteitsmissers kan verminderd worden, moet eerst hun oorzaak doorgrond worden. In dit hoofdstuk wordt de hergebruiksafstand voorgesteld als maat voor lokaliteit van geheugentoegangen. De geheugentoegangen met grote hergebruiksafstand hebben een lage lokaliteit en geven aanleiding tot capaciteitsmissers.

De hergebruiksafstandsdistributie werd gemeten voor de programma's uit de SPEC95FP benchmark, zowel voor als na optimalisatie door een state-of-the-art compiler, die uitgerust is met de meeste programma-optimalisaties die in de laatste decennia werden voorgesteld. Het blijkt dat de compiler een behoorlijk aantal missers met korte hergebruiksafstand kan verwijderen. Het aantal missers met grote hergebruiksafstand wordt echter nauwelijks verkleind. Dit geeft aan dat automatische optimalisatie van missers met grote hergebruiksafstand moeilijk is.

3.1 Terminologie

Definitie 3. *Een* **geheugenreferentie** *correspondeert met een instructie die het geheugen benadert. Een* **geheugentoegang** *is een enkele uitvoering van zo'n referentie. De* **geheugentoegangsstroom** *is de lijst van alle geheugentoegangen die door een programma gemaakt zijn.*

Definitie 4. *Een* **hergebruikspaar** $\langle a_1, a_2 \rangle_L$ *is een paar geheugentoegangen die dezelfde "geheugenlokatie" benaderen, zonder dat er andere geheugentoegangen tussen zitten naar dezelfde lokatie. De geheugenlokatie is de geheugen-*

lijn met lengte L die door de toegang wordt benaderd.

Wanneer $L = 1$, wordt enkel de temporele lokaliteit gemeten. Indien L groter is wordt ook de spatiale lokaliteit die uitgebuit wordt door een cache met lijnlengte L, gevat in de hergebruiksafstand.

Definitie 5. *De data die aangesproken wordt tussen toegangen a_1 en a_2 die een hergebruikspaar $\langle a_1, a_2 \rangle_L$ vormen, wordt voorgesteld door $\mathsf{ADS}\langle a_1, a_2 \rangle_L$ (ADS=Accessed Data Set).*

Definitie 6. *De* **hergebruiksafstand** *van een paar $\langle a_1, a_2 \rangle_L$ is het aantal verschillende geheugenlokaties die aangesproken wordt tussen a_1 en a_2. Het wordt voorgesteld door $\mathsf{RD}(\langle a_1, a_2 \rangle_L)$, en is gelijk aan $|\mathsf{ADS}\langle a_1, a_2 \rangle_L|$.*

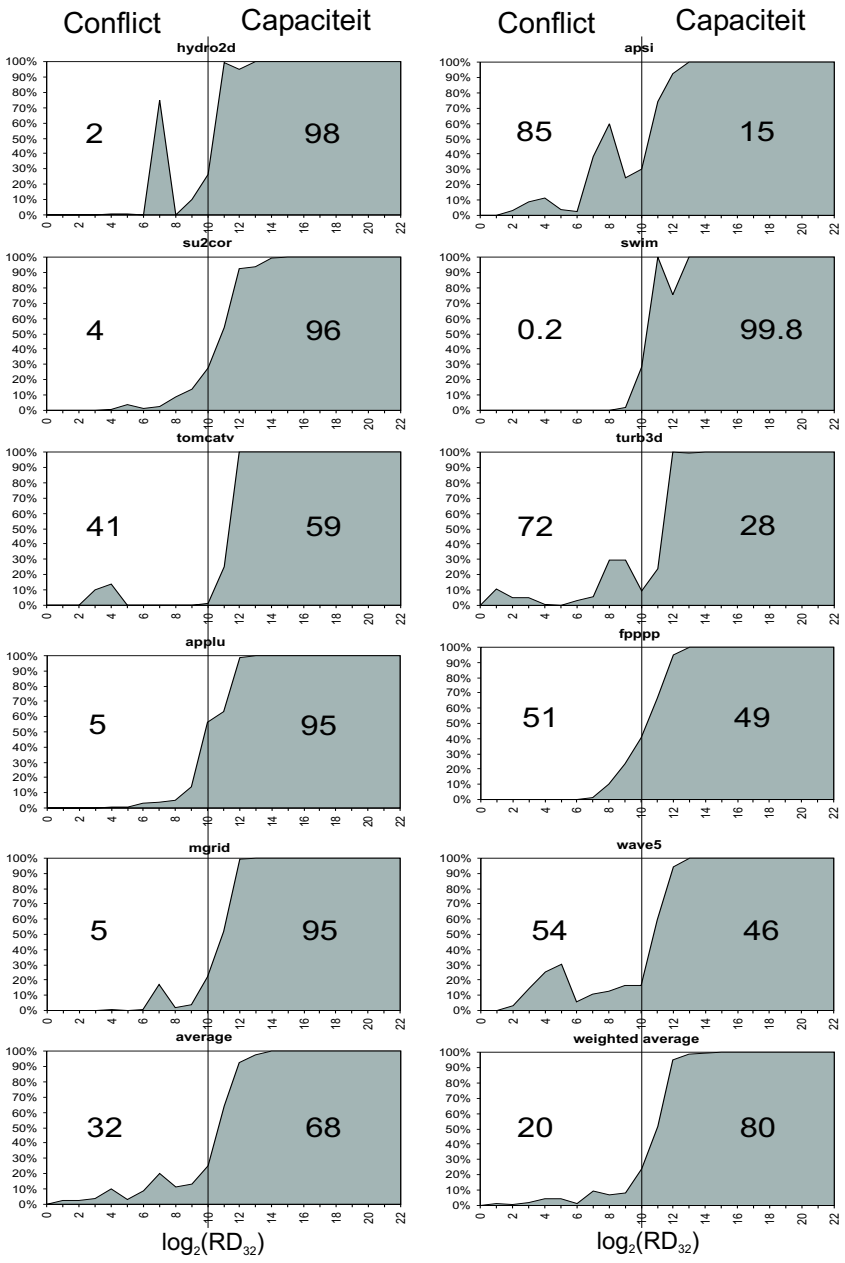
Definitie 7. *Beschouw de hergebruiksparen $\langle a_1, a_2 \rangle_L$ en $\langle a_2, a_3 \rangle_L$. De* **voorwaartse hergebruiksafstand** *van geheugentoegang a_2 is gelijk aan de hergebruiksafstand van $\langle a_2, a_3 \rangle_L$. Als er zo geen paar bestaat, is de voorwaartse hergebruiksafstand ∞. De* **achterwaartse hergebruiksafstand** *van a_2 is gelijk aan de hergebruiksafstand van $\langle a_1, a_2 \rangle_L$. Indien er zo geen paar is, is de achterwaartse hergebruiksafstand oneindig groot.*

Compilers werken met een statische voorstelling van programma's, i.e. met geheugenreferenties i.p.v. geheugentoegangen. Daarom is het ook nuttig om de hergebruiksafstanden per referentie te beschouwen:

Definitie 8. *De* **hergebruiksafstandsdistributie** $\mathsf{RDD}_L(r, s)$ *van een paar referenties r, s is de distributie van de hergebruiksafstanden van alle hergebruiksparen waarbij de eerste toegang veroorzaakt werd door referentie r en de tweede toegang door s. De* **achterwaartse hergebruiksafstandsdistributie van een referentie** *r is de distributie van de achterwaartse hergebruiksafstanden van r, en wordt genoteerd als $\mathsf{BRDD}_L(r)$. De* **voorwaartse hergebruiksafstandsdistributie** *van r is de distributie van de voorwaartse hergebruiksafstanden van de toegangen voortgebracht door r en wordt genoteerd als $\mathsf{FRDD}_L(r)$.*

Theorema 1. Hergebruiksafstandstheorema *In een volledig-associatieve LRU cache met n lijnen zal een toegang een cachetreffer veroorzaken als en slechts als $\mathsf{BRD}_{\mathsf{L_s}}(a) < n$. De geheugenlijn zal in de cache blijven tot het volgende gebruik als en slechts als $\mathsf{FRD}_{\mathsf{L_s}}(a) < n$.*

Het meest gebruikt model om de oorzaak van cachemissers te verklaren is het 3C's model, dat missers classificeert als koude (cold), conflict- of capaciteitsmissers. Die missers kunnen als volgt geclassificeerd worden aan de hand van de achterwaartse hergebruiksafstand.



Figuur 3.1: Miss rate in functie van de hergebruiksafstand voor SPEC95FP. De gesimuleerd cache is een 32KB direct mapped cache met 32 bytes-lange cachelijnen. De missers met een hergebruiksafstand kleiner dan 2^{10} zijn conflictmissers, de andere zijn capaciteitsmissers. Het percentage conflict- en capaciteitsmissers is resp. in de linker en rechter helft van iedere grafiek te zien.

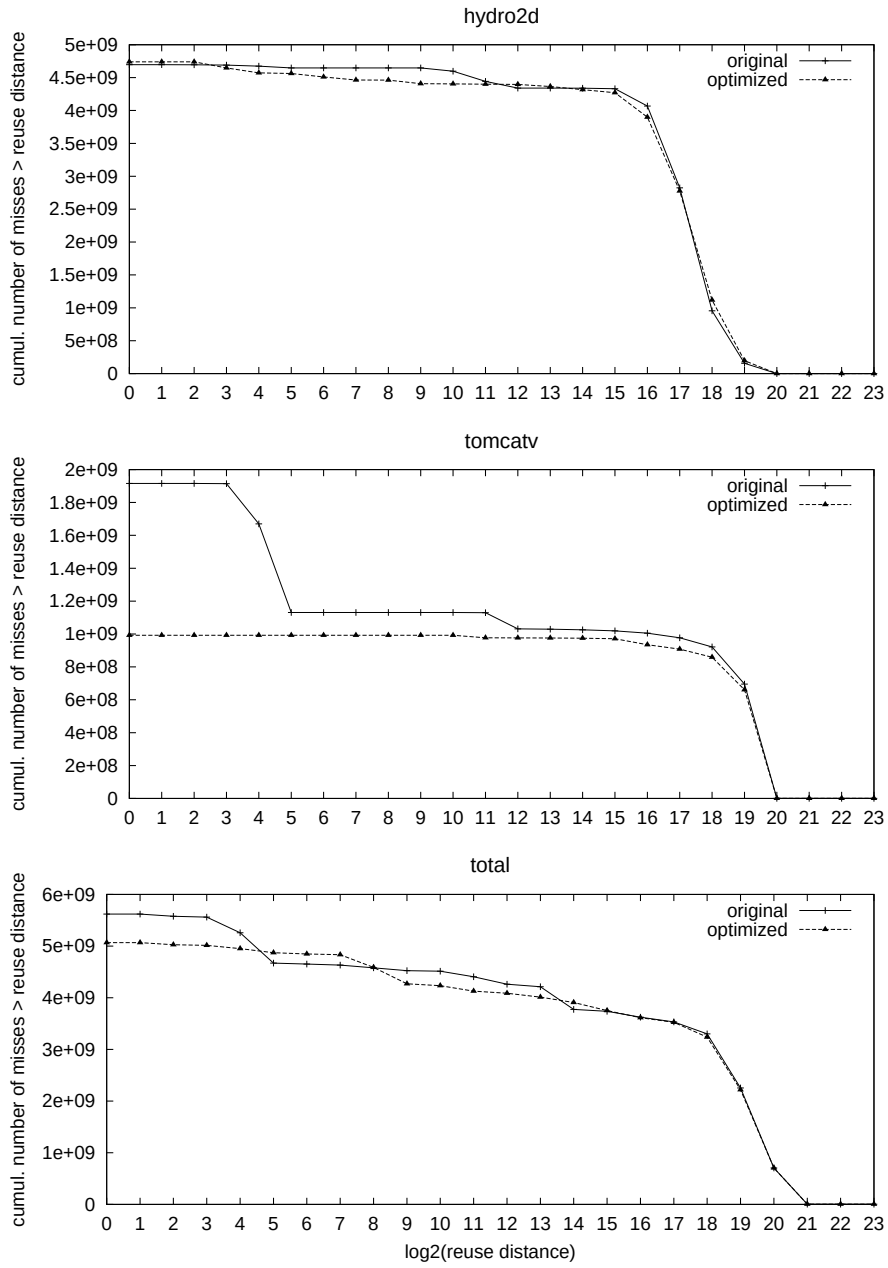
Definitie 9. *Een toegang a die aanleiding geeft tot een cachemisser veroorzaakt*

- *een* **koude misser** *indien* $\mathsf{BRD}(a) = \infty$
- *een* **conflictmisser** *indien* $\mathsf{BRD}(a) < \mathsf{C_s}$
- *een* **capaciteitsmisser** *indien* $\mathsf{C_s} \leq \mathsf{BRD}(a) < \infty$.

3.2 Invloed van Compileroptimalisaties op Hergebruiksafstand en Cachemissers

De programma's in de SPEC95FP benchmark werden geïnstrumenteerd, zodat de hergebruiksafstandsdistributie tijdens de uitvoering wordt opgemeten. Gelijktijdig werd een direct mapped 32KB cache gesimuleerd, en in figuur 3.1 is het percentage cachemissers te zien, in functie van de hergebruiksafstand. Voor een volledig-associatieve cache is deze kans 0 indien de hergebruiksafstand kleiner is dan de cachegrootte, en 1 indien ze groter is. Voor de direct mapped cache toont de figuur dat de overgang van probabiliteit 0 naar 1 rond de cachegrootte geleidelijker gebeurt. De hergebruiksafstand blijft echter een redelijk goede voorspeller voor het cachegedrag. Indien de hergebruiksafstand kleiner is dan de cachegrootte, is de kans op een treffer groot. Indien de hergebruiksafstand veel groter is dan de cachegrootte, is het bijna zeker een misser.

Bovendien werd SGI's Pro64 compiler gebruikt om het cachegedrag van deze programma's zoveel mogelijk te optimaliseren. Het aantal missers zowel voor als na de optimalisatie werd uitgetekend in figuur 3.2. De figuur toont de reverse cumulatieve frequentie van het aantal missers. Enkel hydro2d en tomcatv worden getoond, aangezien die het meest uiteenlopend gedrag vertonen van de volledige programma-verzameling. In het totaaloverzicht is te zien dat er een behoorlijk aantal missers op korte afstand (=conflictmissers) verwijderd werd, maar dat het aantal missers op grote afstand nauwelijks verandert. Die grafiek toont ook dat de meeste missers een zeer grote hergebruiksafstand hebben ($> 2^{17}$). 30% van de conflictmissers werd verwijderd, terwijl slechts 1.2% van de capaciteitsmissers verwijderd werd. Hieruit blijkt dat zelfs de meest geavanceerde compiler-methodes voor het wegwerken van cachemissers slechts in zeer geringe mate missers op grote hergebruiksafstand kunnen wegwerken.



Figuur 3.2: Het aantal cachemissers voor en na optimalisatie, in functie van de hergebruiksafstand. Elk punt in de curve op coördinaat d toont het aantal missers met hergebruiksafstand $\geq d$.

3.3 Samenvatting

In dit hoofdstuk werd de hergebruiksafstand geïntroduceerd als metriek voor lokaliteit. Voor volledig-associatieve caches kan het cachegedrag perfect voorspeld worden aan de hand van de hergebruiksafstand. Ook voor minder associatieve caches kan het cachegedrag goed voorspeld worden aan de hand van de hergebruiksafstand.

Er blijkt verder dat de bestaande automatische programma-optimalisaties om het cachegedrag te verbeteren nauwelijks enige missers op grote hergebruiksafstand kunnen wegwerken, terwijl de meeste missers net een zeer grote hergebruiksafstand hebben.

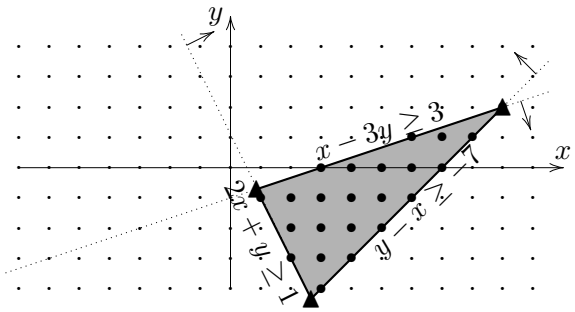
Hoofdstuk 4

Hergebruiksafstands-vergelijkingen

Hergebruiksafstandsvergelijkingen beschrijven hergebruiksparen, de lokaties die aangesproken worden tussen gebruik en hergebruik, en de bijhorende hergebruiksafstand. De oplossingen van deze vergelijkingen zijn multivariate Ehrhart-polynomen, waarbij de variabelen bestaan uit de lusvariabelen en de programmaparameters. De vergelijkingen maken het tijdrovende profileren overbodig. Een belangrijker voordeel van de vergelijkingen is dat de resulterende Ehrhart-polynomen de hergebruiksafstand voor alle mogelijke programma-uitvoeringen beschrijven, terwijl de geprofileerde hergebruiksafstandsdistributies die slechts voor een specifieke opgemeten programma-uitvoering beschrijven.

Om efficiënte compileroptimalisaties toe te laten moet de lokaliteit van het programma in relatief compacte vorm worden voorgesteld. Bij profilering wordt dit bereikt door slechts de distributie van de hergebruiksafstanden bij te houden per instructie, waardoor je geen precieze informatie meer hebt over de lokaliteit van iedere individuele geheugentoegang. De vergelijkingen bieden dit wel: de Ehrhart-polynomen bieden een compacte voorstelling van de lokaliteit van iedere individuele geheugentoegang.

De vergelijkingen zijn gebaseerd op polytopen en Presburger-formules. Deze worden besproken in de eerste sectie. In de tweede sectie worden de hergebruiksafstandsvergelijkingen voorgesteld. De daaropvolgende secties gaan kort in op het oplossen van de vergelijkingen.



Figuur 4.1: Grafische voorstelling van de polytoop die bepaald wordt door $2x+y \geq 1 \wedge x-3y \geq 3 \wedge y-x \geq -7$. De drie hoekpunten zijn aangeduid door ▲'s op coördinaten $(\frac{6}{7}, \frac{-5}{7})$, $(\frac{8}{3}, \frac{-13}{3})$ en $(9, 2)$.

4.1 Polyhedraal Programmamodel

De analytische beschrijving van hergebruiksafstanden vereist een raamwerk waarin programmaeigenschappen zoals uitvoeringsvolgorde en de aangesproken data tussen twee tijdstippen kan gemodelleerd worden. In dit werk worden deze eigenschappen beschreven aan de hand van polytopen in Presburger-formules.

4.1.1 Polytopen en Presburger-formules

Definitie 10. *Een verzameling van vectoren P in $\mathbb{Z}^n$ worden een* **geheeltallig polyhedron** *genoemd indien*

$$P = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} \geq \mathbf{b}\}, \tag{4.1}$$

waarbij A een matrix van gehele getallen en $\mathbf{b}$ een vector van gehele getallen is.

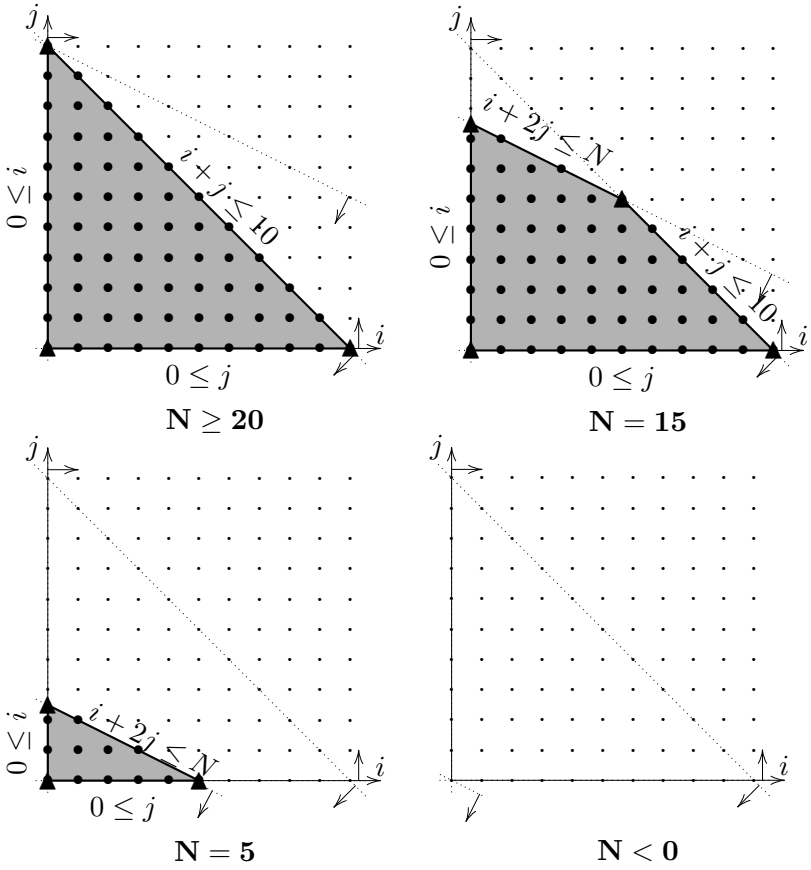
Een geheeltallig polyhedron dat een eindig aantal vectoren bevat wordt een **geheeltallig polytoop** *genoemd.*

Een voorbeeld van een geheeltallig polytoop is te vinden in figuur 4.1.

Definitie 11. *Een* **geparameteriseerde geheeltallige polytoop** $P_{\mathbf{p}}$ *is een familie van polytopen gedefinieerd door*

$$P_{\mathbf{p}} = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} \geq B\mathbf{p} + \mathbf{b}\}, \mathbf{p} \in \mathbb{Z}^m, \tag{4.2}$$

waarbij A en B geheeltallige matrices zijn, $\mathbf{b}$ een geheeltallige vector is, en $\mathbf{p}$ een vector is die parameters bevat.



Figuur 4.2: De grafische voorstelling van de geparameteriseerde polytoop gedefinieerd door $\{(i,j) \in \mathbb{Z}^2 : 0 \leq i,j \wedge i+j \leq 10 \wedge i+2j \leq N\}$. Er is een enkele parameter N. Het aantal hoekpunten is afhankelijk van de parameterwaarde N.

Voorbeeld 1. *In figuur 4.2 wordt de volgende geparameteriseerde polytoop grafisch voorgesteld voor een aantal verschillende parameter-waarden:*

$$\begin{aligned} P_{\mathbf{N}} &= \{(i,j) \in \mathbb{Z}^2 : 0 \leq i \wedge 0 \leq j \wedge i+j \leq 10 \wedge i+2j \leq N\} \\ &= \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 | \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \\ -1 & -2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ -10 \\ 0 \end{pmatrix} \right\} \end{aligned}$$

Definitie 12. *Een* **Presburger-formule** *is een formule die bestaat uit gehele getallen, gehele variabelen, optelling, vermenigvuldiging met gehele constan-*

```
do i = 1, N
  A(i,i)=i
enddo
do i = 1, N
 A(2i,1)=1
 do j = i+2, N-i
  if (i<>j) A(i,j-i) = A(3+j,i)
 enddo
enddo
```

Figuur 4.3: Voorbeeld van een programma in het polyhedrale model.

ten, de relaties $<$ en $=$, de logische bewerkingen $\vee$, $\wedge$ en $\neg$, en de kwantifactoren $\exists$ en $\forall$. Een voorbeeld van zo'n formule is $\forall x, \exists y : 3x + y = 2 \wedge x < y$.

4.1.2 Programmavoorstelling in het Polyhedraal Model

In het polyhedrale programmamodel worden eigenschappen zoals uitvoeringsvolgorde, iteratieruimten, aangesproken data, e.d. voorgesteld door middel van Presburger-formules en geheeltallige polytopen.

Definitie 13. *The* **verzameling van alle referenties** *in een programma wordt voorgesteld door $\mathcal{R}$. The* **verzameling van de array-variabelen** *wordt voorgesteld door $\mathcal{V}$. De* **iteratieruimte** *van een referentie r wordt voorgesteld door $\mathsf{IS}(r)$. De* **geheugenlokatie** *die aangesproken wordt door referentie r tijdens iteratie i wordt voorgesteld door $r@i$. Het feit dat iteratie i van referentie r* **uitgevoerd wordt voor** *iteratie j van referentie s wordt neergeschreven als $i_r \lessdot j_s$. De verzameling van parametrische programmaconstanten wordt voorgesteld door $\mathcal{P}$.*

Voorbeeld 2. *Om de notaties in definitie 13 te verduidelijken wordt hier een aantal voorbeelden gegeven met betrekking op het programma in figuur 4.3.*

- $\mathcal{V} = \{A\}$.
- $\mathcal{R} = \{\mathtt{A(i,i)}, \mathtt{A(2i,1)}, \mathtt{A(i,j-i)}, \mathtt{A(3+j,i)}\}$.
- *De iteratieruimte* $\mathsf{IS}(\mathtt{A(i,j-i)}) =$
  $\{(i,j) : (1 \leq i \leq N) \wedge (i+2 \leq j \leq N-i) \wedge \neg(i=j)\}$
- *Het array-element aangesproken door* `A(3+j,i)` *tijdens iteratie (i=3, j=6) is* $\mathtt{A(3+j,i)}@(i=3, j=6) = A(9,3)$.

- *De volgorde* $(i)_{\mathtt{A(2i,1)}} \lessdot (i', j')_{\mathtt{A(i,j-i)}} = i \leq i'$.
- $\mathcal{P} = \{\mathtt{N}\}$.

4.2 Hergebruiksafstandsvergelijkingen

De hergebruiksafstand wordt berekend in een aantal stappen:

1. Eerst worden de hergebruiksparen in de geheugentoegangsstroom berekend. Voor iedere paar *referenties* (r,s) wordt een Presburger-formule $\mathsf{reuse}\,(r \rightarrow s)$ geconstrueerd, die alle hergebruiken voorstelt waarvoor de eerste toegang gegenereerd werd door r, en de tweede toegang door s.
2. Voor elk paar $\mathsf{reuse}\,(r \rightarrow s)$ wordt er een verzameling van geparameteriseerde polytopen $(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s)))$ geconstrueerd die de data voorstelt die aangesproken wordt tussen gebruik en hergebruik.
3. Het aantal geheeltallige punten in die verzameling polytopen wordt berekend. Het resultaat is een aantal Ehrhart-polynomen, die de hergebruiksafstand beschrijven.

4.2.1 Hergebruikspaar

In de eerste stap worden de hergebruiksparen voorgesteld door middel van een Presburger-formule. Deze formule bestaat uit de volgende onderdelen:

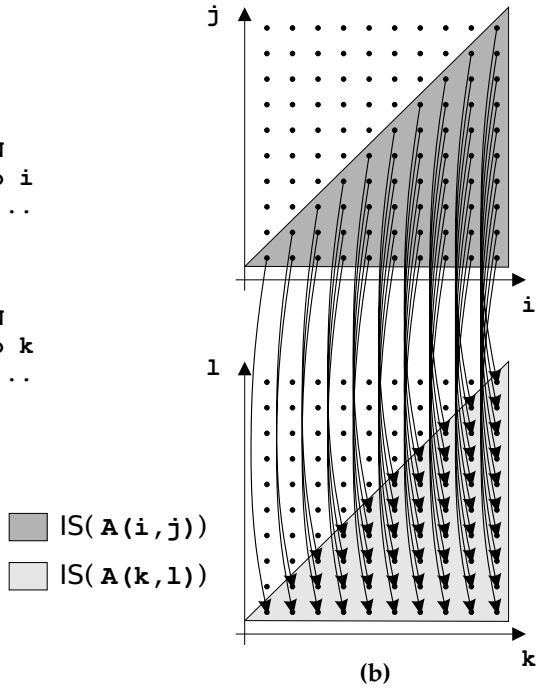
$$\forall r, s \in \mathcal{R} : \mathsf{reuse}\,(r \rightarrow s) = \{(I_r, J_s) \in \mathbb{Z}^n : \text{voorwaarden (4.4a)– (4.4d)}\} \quad (4.3)$$

$I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s)$	(*iteratieruimte*)	(4.4a)
$I_r \lessdot J_s$	(*uitvoeringsvolgorde*)	(4.4b)
$r@I_r = s@J_s$	(*zelfde lokatie*)	(4.4c)
$\forall t \in \mathcal{R} : \neg(\exists K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s \wedge t@K_t = r@I_r)$	(*geen tussenliggend gebruik*)	(4.4d)

```
for i := 1 to N
  for j := 1 to i
    A(i,j) := ...
  endfor
endfor

for k := 1 to N
  for l := 1 to k
    A(k,l) := ...
  endfor
endfor
```

(a)



- reuse ($\mathbf{A(i,j)} \rightarrow \mathbf{A(k,l)}$) =
  $\{(i,j,k,l) : 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge$
  $1 \leq k \leq N \wedge 1 \leq l \leq k \wedge i = k \wedge j = l\}$
- reuse_F ($\mathbf{A(i,j)}$)= $\{(i,j) : 1 \leq i \leq N \wedge 1 \leq j \leq i\}$
- reuse_B ($\mathbf{A(i,j)}$)=$\emptyset$

Figuur 4.4: De hergebruiksparen in een eenvoudig programma. In (a) is de broncode te zien. In (b) worden de hergebruiksparen als pijlen tussen de iteratiepunten van de twee referenties voorgesteld. In (c) zijn die hergebruiksparen als geparameteriseerde geheeltallige polytopen neergeschreven.

De formules hierboven geven de voorwaarden weer waaraan voldaan moet zijn opdat er een hergebruik is tussen $r@I_r$ en $s@J_s$. Voorwaarde (4.4a) bepaalt dat enkel de iteraties binnen de lusgrenzen in aanmerking komen; voorwaarde (4.4b) zegt dat het gebruik voor het hergebruik moet uitgevoerd worden; voorwaarde (4.4c) geeft aan dat er slechts hergebruik is als beide toegangen dezelfde geheugenlokatie aanspreken; en voorwaarde (4.4d) toont dat er geen andere toegangen tussen gebruik en hergebruik mogen zitten die dezelfde lokatie aanspreken. De iteratiepunten waarvoor er respectievelijk voorwaarts en achterwaarts hergebruik is voor referentie s wordt door de volgende formules beschreven:

$$\begin{aligned} \mathsf{reuse_F}\,(s) &= \bigcup_{\forall t \in \mathcal{R}} \{I_s : \exists J_t \in \mathsf{IS}(t) : (I_s, J_t) \in \mathsf{reuse}\,(s \rightarrow t)\} \\ \mathsf{reuse_B}\,(s) &= \bigcup_{\forall r \in \mathcal{R}} \{J_s : \exists I_r \in \mathsf{IS}(r) : (I_r, J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \end{aligned} \tag{4.5}$$

Een voorbeeldje van deze formules voor een eenvoudig programma is te zien in figuur 4.4.

4.2.2 Aangesproken Data tussen Hergebruiken

De functie map_r beeldt een iteratieruimte af op de geheugenlokaties die aangesproken worden door een referentie r in die iteratieruimte. De functie $\mathsf{iters}_r(I_r, J_s)$ is de verzameling van iteraties van t die uitgevoerd worden tussen iteratie I_r en iteratie J_s:

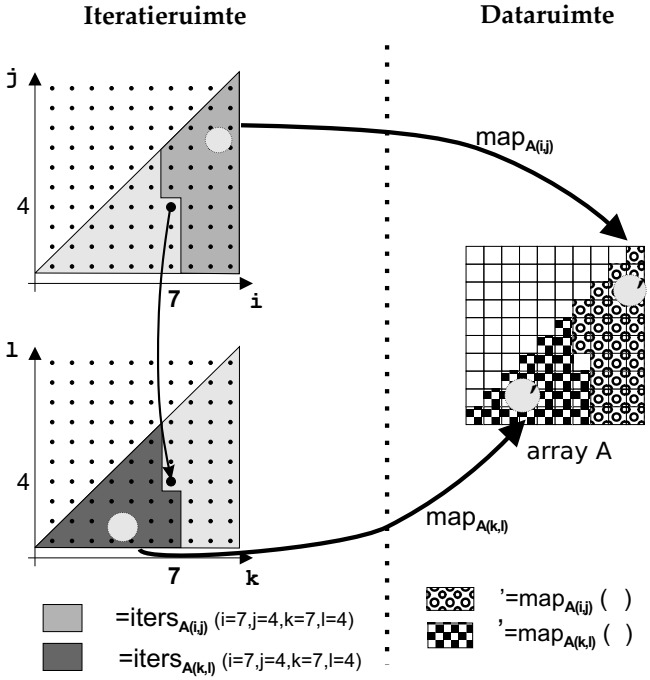
$$\mathsf{map}_r = \{I \rightarrow r@I : I \in \mathsf{IS}(r)\} \tag{4.6}$$

$$\mathsf{iters}_t(I_r, J_s) = \{(I_r, J_s) \rightarrow K_t : K_t \in \mathsf{IS}(t) \wedge I_r \lessdot K_t \lessdot J_s\} \tag{4.7}$$

De data die aangesproken wordt tussen gebruik en hergebruik van het geparameteriseerde hergebruikspaar $\mathsf{reuse}\,(r \rightarrow s)$, wordt voorgesteld door $\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))$. (ADS=Accessed Data Set):

$$\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s)) \;=\; \bigcup_{t \in \mathcal{R}} \mathsf{map}_t(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))), \tag{4.8}$$

Vergelijking (4.8) duidt aan dat de aangesproken data tussen gebruik en hergebruik berekend wordt door alle iteraties tussen gebruik

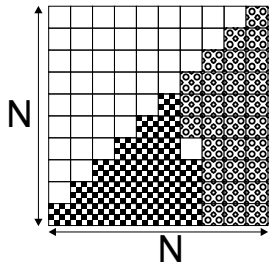


(a)

$$
\begin{aligned}
&\mathsf{ADS}\,(\mathsf{reuse}\,(\texttt{A(i,j)} \rightarrow \texttt{A(k,l)})) = \\
&\big\{(x,y) : \big(1 \le i \le N \wedge 1 \le j \le i \wedge && \mathsf{IS}(\mathtt{A}(\mathtt{i},\mathtt{j})) \\
&\qquad 1 \le k \le N \wedge 1 \le l \le k \wedge && \mathsf{IS}(\mathtt{A}(\mathtt{k},\mathtt{l})) \\
&\qquad i = k \ \wedge \ j = l\big) \wedge && \textit{zelfde lokatie} \\
&\qquad \big(0 \le x < i \ \vee \ x = i \wedge y < j \vee && \textit{aangesproken data tussen} \\
&\qquad\ N \ge x > i \ \vee \ x = i \wedge y > j\big)\big\} && \textit{gebruik en hergebruik}
\end{aligned}
$$

(b)

Figuur 4.5: Berekening van $\mathsf{ADS}(\mathsf{reuse}\,(\texttt{A(i,j)} \rightarrow \texttt{A(k,l)}))$. Er wordt een enkel hergebruikspaar getoond, van referentie `A(i,j)` in iteratie $i = 7, j = 4$) naar referentie `A(k,l)` voor iteratie (k=7, l=4). Op de linkerzijde is $\alpha = \mathsf{iters}_{\mathtt{A(i,j)}}(i{=}7, j{=}4, k{=}7, l{=}4)$ en $\beta = \mathsf{iters}_{\mathtt{A(k,l)}}(i{=}7, j{=}4, k{=}7, l{=}4)$ aangeduid in de iteratieruimte. De functies $\mathsf{map}_{\mathtt{A(i,j)}}$ en $\mathsf{map}_{\mathtt{A(k,l)}}$ beelden de iteratieruimtes α en β af op de overeenkomstige dataruimte α' en β'.



(a)

- $\mathsf{RD}(\texttt{A(i,j)}, \texttt{A(k,l)})$
  $= \mathcal{E}\,(\mathsf{ADS}\,(\texttt{A(i,j)}, \texttt{A(k,l)})\ ;\ \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l}, \texttt{N})$
  $= \dfrac{N^2+N}{2} - 1$
- $\mathsf{FRD}(\texttt{A}(\texttt{i}, \texttt{j})) = \dfrac{N^2+N}{2} - 1$
- $\mathsf{BRD}(\texttt{A}(\texttt{k}, \texttt{l})) = \dfrac{N^2+N}{2} - 1$
- $\mathsf{COLDM}(\texttt{A}(\texttt{i}, \texttt{j})) = \{(i,j) : 1 \leq i \leq N \wedge 1 \leq j \leq i\}$
- $\mathsf{NOKEEP}(\texttt{A}(\texttt{i}, \texttt{j})) = \{(i,j) : 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge \dfrac{N^2+N}{2} - 1 \geq \mathsf{C_s}\}$
- $\mathsf{CAPM}(\texttt{A}(\texttt{k}, \texttt{l})) = \{(k,l) : 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge \dfrac{N^2+N}{2} - 1 \geq \mathsf{C_s}\}$

(b)

Figuur 4.6: In **(a)** wordt de data die aangesproken wordt tussen het gebruik en hergebruik in figuur 4.5 getoond. De hoeveelheid data hierin is $\frac{N^2+N}{2} - 1$. In **(b)** worden de vergelijkingen (4.9)–(4.14) toegepast op dit voorbeeld.

en hergebruik te bepalen, waarna de datagebieden berekend worden die tijdens die iteraties worden aangesproken. Een voorbeeld hiervan is te zien in figuur 4.5.

4.2.3 Hergebruiksafstand van een Hergebruikspaar

De hergebruiksafstand van een hergebruikspaar is het aantal verschillende geheeltallige punten in $\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))$:

$$\mathsf{RD}(r,s) = \mathcal{E}(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))\,; I_r, J_s, \mathcal{P}), \tag{4.9}$$

waarbij $\mathcal{E}(P;p)$ het aantal geheeltallige punten in de verzameling polytopen P voorstelt en p als parameters geïnterpreteerd moeten worden. De voorwaartse en achterwaartse hergebruiksafstand van een referentie kan dan als volgt berekend worden:

$$\mathsf{FRD}(r) = \sum_{s \in \mathcal{R}} \mathcal{E}(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))\,; I_r, \mathcal{P}) \tag{4.10}$$

$$\mathsf{BRD}(s) = \sum_{r \in \mathcal{R}} \mathcal{E}(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))\,; J_s, \mathcal{P}) \tag{4.11}$$

De iteratiepunten waar koude missers optreden, zijn die punten waar data voor het eerste aangesproken wordt:

$$\mathsf{COLDM}(r) = \{I : I \in \mathsf{IS}(r) \wedge I \notin \mathsf{reuse}_{\mathsf{B}}\,(r)\} \tag{4.12}$$

De iteratiepunten waar capaciteitsmissers optreden zijn diegene waar de achterwaartse hergebruiksafstand groter is dan de cachegrootte:

$$\mathsf{CAPM}(r) = \{I : \mathsf{BRD}(r) \geq \mathsf{C_s} \wedge I \in \mathsf{reuse}_{\mathsf{B}}\,(r)\}, \tag{4.13}$$

De iteratiepunten waarvoor de data niet in de cache bewaard zal blijven tot de volgende toegang naar die data wordt als volgt beschreven:

$$\mathsf{NOKEEP}(r) = \{I : \mathsf{FRD}(r) \geq \mathsf{C_s} \wedge I \in \mathsf{reuse}_{\mathsf{F}}\,(r)\}, \tag{4.14}$$

Een voorbeeld van deze formules is te vinden in figuur 4.6.

4.3 Enumeratie van Geparameteriseerde Polytopen

Een van de belangrijkste stappen bij het oplossen van de hergebruiksafstandsvergelijkingen is het bepalen van het aantal geheeltallige punten in een verzameling van geparameteriseerde polytopen, zie vergelijkingen (4.9)- (4.11).

Ehrhart [65] en Clauss [50] toonden aan dat de vorm van de oplossing afhankelijk is van de geparameteriseerde hoekpunten van de polytoop. Bovendien zijn niet alle hoekpunten gedefinieerd voor alle parameterwaarden. Het domein van alle parameterwaarden kan gepartitioneerd worden zodat de geparameteriseerde hoekpunten telkens over een volledige partitie gedefinieerd zijn. Bij iedere partitie hoort dan een zogenaamde Ehrhart-polynoom die het aantal geheeltallige punten uitdrukt in functie van de parameters.

Voorbeeld 3. *Beschouw de geparameteriseerde polytoop in figuur 4.2. Het domein van de parameterwaarden kan opgesplitst worden als volgt, met bijhorende parametrische hoekpunten:*

Geldigheidsdomein	*Parametrische hoekpunten*
$N < 0$	$\emptyset$
$0 \leq N \leq 10$	$(0,0), (N,0), (0, \frac{N}{2})$
$10 \leq N \leq 20$	$(0,0), (0, \frac{N}{2}), (20-N, N-10), (10,0)$
$20 \leq N$	$(0,0), (10,0), (0,10)$

Het aantal punten in dit polytoop is

$$\mathcal{E}(P;N) = \begin{cases} 0 & \textit{in domein} \quad N \leq 0 \\ \frac{N^2}{4} + N + 1 - \frac{N \bmod 2}{4} & \textit{in domein} \quad 0 \leq N \leq 10 \\ -\frac{N^2}{4} + \frac{21N}{2} - 44 - \frac{N \bmod 2}{4} & \textit{in domein} \quad 10 \leq N \leq 20 \\ 66 & \textit{in domein} \quad 20 \leq N \end{cases}$$

De oplossingsmethodes om de Ehrhart-polynomen horende bij de geldigheidsdomeinen te berekenen zijn te complex om in deze samenvatting in detail te bespreken. De Engelsetalige tekst bevat een beschrijving van twee methodes: een reeds bekende methode, die echter in een aantal gevallen faalt, en een nieuwe methode die altijd slaagt in het berekenen van de Ehrhart-polynomen. Voorts dienen voor het oplossen van de hergebruiksafstandsvergelijkingen niet enkel polytopen geteld te worden, maar ook *verzamelingen* van polytopen, wat extra moeilijkheden meebrengt. Indien er polytopen in die verzameling overlappen, moet er namelijk voor gezorgd worden dat de punten in de doorsnede slechts 1 maal geteld worden. Algoritmen om deze problemen aan te pakken worden ook in detail besproken in de Engelstalige tekst.

Een aantal voorbeelden van voorwaartse hergebruiksafstanden is te vinden in appendix A, op pagina 231.

4.4 Samenvatting

In dit hoofdstuk werden de hergebruiksafstandsvergelijkingen voorgesteld. Deze laten toe om de hergebruiksafstanden voor programma's in het polyhedrale model te beschrijven voor alle mogelijke uitvoeringen.

Bovendien wordt de hergebruiksafstand voor alle geheugentoegangen in een compacte vorm beschreven door een verzameling van Ehrhart-polynomen. Het oplossen van de vergelijkingen vereist een methode voor het berekenen van het aantal geheeltallige punten in geparameteriseerde polytopen. Er werd een nieuwe methode voorgesteld om die oplossing te bepalen, die in tegenstelling tot de reeds bekende methode, altijd de oplossing kan berekenen.

In tegenstelling tot vroegere voorstellen om cachegedrag en lokaliteit te berekenen zonder over te gaan tot profilering, geven hergebruiksafstandsvergelijkingen exacte oplossingen voor alle programma's in het polyhedraal model, en beschrijven ze de lokaliteit voor iedere individuele geheugentoegang. Bovendien kan de methode werken met symbolische lusgrenzen en arraygroottes.

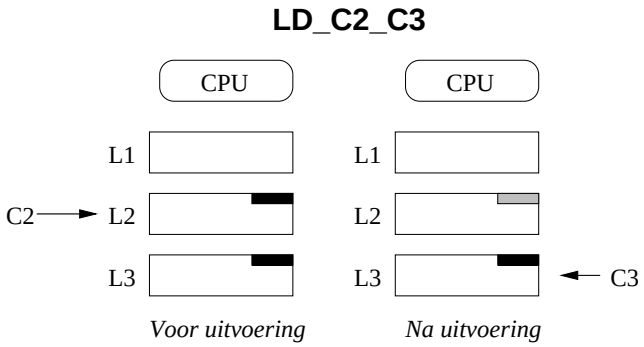
Hoofdstuk 5

Cache Hint-Selectie

Cache hints zijn annotaties aan geheugeninstructies. Ze komen voor in twee verschillende gedaantes: bronhints *en* doelhints*. De bronhint duidt het snelste cacheniveau aan waar de data zich vermoedelijk bevindt, terwijl de doelhint het snelste cacheniveau aanduidt waar de data bewaard moet blijven. In dit hoofdstuk worden beide soorten hints gegenereerd door een compiler, gebaseerd op de hergebruiksafstand. Enerzijds worden er* statische *hints gegenereerd, gebaseerd op een hergebruiksafstandsdistributie. Anderzijds worden er* dynamische *hints aangemaakt op basis van de polynomen die berekend werden uit de hergebruiksafstandsvergelijkingen. De statische hints bepalen een enkele hint per instructie, terwijl de dynamische hints per geheugentoegang bepaald worden.*

5.1 Cache Hints in EPIC Architecturen

De meeste recente instructie-sets definiëren cache-instructies zoals prefetches. Bovendien kan je bij de meeste van die instructie-sets bij prefetches door middel van cache hints bepalen in welk cacheniveau de data bewaard moet blijven. Daarentegen kan in de meeste architecturen geen informatie aan load- of store-instructies gehangen worden omtrent het cachegedrag van die instructie. Bij EPIC (=Explicitly Parallel Instruction Computing) architecturen, zoals HP's HPL-PD architectuur en Intel's IA-64, is dit echter wel mogelijk. Aan iedere instructie kunnen twee cache hints gehangen worden: een *bronhint* en een *doelhint*. Een voorbeeld van een load-instructie uit de HPL_PD architectuur met twee hints is te vinden in figuur 5.1.

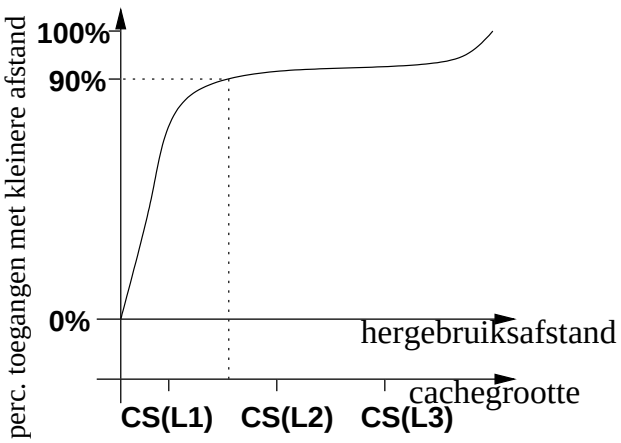


Figuur 5.1: Effect van de cache hints in de load-instructie `LD_C2_C3`. De bronhint `C2` in de instructie duidt aan dat de opgevraagde data waarschijnlijk in de L2-cache kan gevonden worden. De doelhint `C3` bepaalt dat de data in de cache-hiërarchie niet hoger dan de L3-cache moet bewaard worden. Bijgevolg wordt de data het eerste slachtoffer wanneer er data uit de L2-cache verwijderd moet worden.

Traditionele compilers veronderstellen dat alle geheugeninstructies hun data vinden in de L1-cache. Bijgevolg wordt er verondersteld dat alle geheugeninstructies de latentie van de L1-cache hebben. Indien dat niet zo is, kunnen afhankelijke instructies door de compiler te dicht bij elkaar geplaatst zijn, waardoor de data nog niet geladen is op het moment dat de instructie gestart moet worden. Met behulp van de bronhints kan de compiler weten dat een grotere latentie verondersteld moet worden. Hierdoor zal de instructiescheduler in de compiler trachten voldoende parallelle instructies tussen de geheugeninstructie en zijn afhankelijke instructies te plaatsen, om de processor tijdens het laden van de data nuttig werk te laten doen.

De doelhints worden gecommuniceerd naar de processor. Door de doelhints zorgvuldig te selecteren wordt data slechts bijgehouden in de cacheniveaus waar ze zullen blijven tot het volgende hergebruik, wat cachevervuiling in de kleinere cacheniveaus tegengaat.

In de IA-64 architectuur worden bronhints niet expliciet gedefinieerd, omdat ze niet naar de processor gecommuniceerd moeten worden. De IA-64 biedt een variant aan van de doelhints `C1`, `C2`, `C3`, `C4`, en die worden neergeschreven als respectievelijk `.t1`, `.nt1`, `.nt2`, `.nta`.



Figuur 5.2: Een mogelijke cumulatieve hergebruiksafstandsdistributie voor een instructie. Het cacheniveau waarvoor 90% van de toegangen een hergebruiksafstand kleiner dan de cachegrootte hebben is L2.

5.2 Statische Hint Selectie

5.2.1 Cache Hint per Toegang

De *bronhint* duidt het snelste cacheniveau aan waar de data gevonden kan worden, terwijl de *doelhint* het snelste cacheniveau aanduidt waar de data het best bewaard blijft. Deze cacheniveau's kunnen als volgt bepaald worden aan de hand van de hergebruiksafstand.

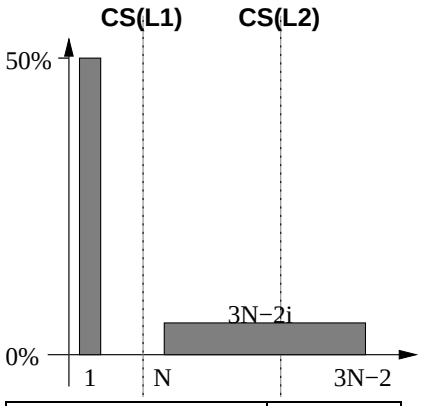
Bronhintselectie De bronhint horende bij toegang t wordt gekozen, zo dat die het snelste cacheniveau aanduidt waarvoor $\mathsf{BRD}(t) \leq \mathsf{C_s}$. Dit is immers het snelste cacheniveau waar de data gevonden zal worden.

Doelhintselectie De doelhint horende bij toegang t wordt gekozen, zo dat die het snelste cacheniveau aanduidt waarvoor $\mathsf{FRD}(t) \leq \mathsf{C_s}$. Dit is immers het snelste cacheniveau waarvoor de data bewaard zal blijven tot het volgende hergebruik.

5.2.2 Cache Hint per Instructie

Een cache hint wordt geannoteerd aan een *instructie*, en niet aan een enkele toegang. Bijgevolg kan er voor alle toegangen die door een instructie worden gegenereerd, slechts een vaste bron- en doelhint gegeven worden. Die vaste (statische) hints worden als volgt bepaald.

```
do i=1,N
  do j=1,2
    B(i,j) = A(i)
  enddo
enddo
do j=1,N
  A(j)=0
enddo
```



iteratiedomein	FRD
$1 \leq i \leq N \wedge j = 1$	1
$1 \leq i \leq N \wedge j = 2$	$3N - 2i$

Figuur 5.3: Een voorbeeld van een hergebruiksafstandsdistributie waarvoor het niet mogelijk is om een enkele hint te selecteren die goed is voor alle toegangen. Links wordt een programma getoond, terwijl rechts de hergebruiksafstandsdistributie van referentie `A(i)` wordt getoond. De helft van alle toegangen hebben hergebruiksafstand 1, waarbij cache hint C1 hoort. De andere helft heeft hergebruiksafstand $3N - 2i$, waarbij een cache hint hoort die afhangt van de waarden van N en i.

Per instructie wordt zowel de voorwaartse als de achterwaartse hergebruiksafstandsdistributie opgemeten. Een voorbeeld van zo'n distributie is te vinden in figuur 5.2. Uit die figuur blijkt dat het onmogelijk is om een enkele hint te selecteren die ideaal is voor alle toegangen. Daarom wordt de bronhint zo gekozen dat voor de 90% van alle toegangen, de data zeker in dat cacheniveau te vinden is. De doelhint wordt zo gekozen dat voor ten minste 90% van alle toegangen de data in het snelste cacheniveau zal geplaatst worden waarin het zal gevonden worden tijdens het volgende hergebruik.

5.3 Dynamisch Hint Selectie

De statische hint selectie uit sectie 5.2 heeft twee belangrijke beperkingen. Ten eerste er is slechts een hint per instructie, waardoor de hint niet geschikt gemaakt kan worden voor alle toegangen. Ten tweede wordt de hint geselecteerd op basis van een enkele uitvoering van

het programma. Bij een andere uitvoering, met andere input, kan de lokaliteit, de hergebruiksafstandsdistributie, en het cachegedrag van de instructie sterk veranderen. Een voorbeeld van een instructie waarbij beide problemen optreden is te zien in figuur 5.3.

5.3.1 Codegeneratie

Om dynamische doelhints mogelijk te maken dient de instructie set lichtjes aangepast te worden. Bij de geheugeninstructies zoals load, store en prefetch dient er een extra inputregister bijgevoegd te worden, die de voorwaartse hergebruiksafstand van de huidige geheugentoegang bevat. Een voorbeeld van dergelijke load-instructie is `ld r5=[r6],frd=r7`, waarbij `r5` het register is waarin de data geladen wordt, `r6` het register is dat het adres bevat, en `r7` het extra inputregister is dat de voorwaartse hergebruiksafstand bevat. De hergebruiksafstand die in `r7` terecht moet komen kan berekend worden door de juiste Ehrhart-polynoom te evalueren die bekomen werd na het oplossen van de hergebruiksafstandsvergelijkingen.

Een bijkomend voordeel van deze manier van cache hint selectie, is dat at compile time de cachegroottes niet gekend hoeven te zijn. Op die manier is het mogelijk om tijdens een compilatie de cache hints te optimaliseren voor alle processors die dezelfde instructieset ondersteunen, ongeacht de grootte van de caches. Bovendien is het ook eenvoudig om code te genereren zonder cache hints. Op de meeste architecturen bevat register `r0` immers altijd de waarde 0, en de instructie `ld r5=[r6],frd=r0` duidt dus aan dat de data in alle cacheniveau's bewaard moet blijven. Bijgevolg wordt het standaard LRU-vervangingsalgoritme gebruikt voor die data.

5.3.2 Overheadreductie

Het berekenen van de voorwaartse hergebruiksafstand voor iedere geheugentoegang kan aanleiding geven tot grote overheads. Er moet immers voor iedere toegang met een aantal if-testen bepaald worden in welk geldigheidsdomein het huidige iteratiepunt valt, wat aanleiding kan geven tot een veel complexer controle-verloop. Deze overhead wordt gereduceerd door enkel de dominante domeinen (zie sectie 4.6 in de Engelstalige tekst) te beschouwen. Dit zorgt ervoor dat bij iedere referentie slechts een enkele polynoom hoort, en dat er dus geen extra

```
do i=1,N
 do j=1,2
 if (j.eq.1) then
  FRD_Ai = 1
 if (j.eq.2) then
  FRD_Ai = 3*N-2*i
 B(i,j) = A(i)
 enddo
enddo
do j=1,N
 A(j)=0
enddo
```

(a) exacte doelhints

```
do i=1,N
 FRD_Ai = 1
 B(i,1) = A(i)
 FRD_Ai = 3*N-2*i
 B(i,2) = A(i)
enddo
do j=1,N
  A(j)=0
enddo
```

(b) na loop peeling

```
FRD_A1 = 1
FRD_A2 = 3*N-2
do i=1,N
 B(i,1) = A(i), frd=FRD_A1
 B(i,2) = A(i), frd=FRD_A2
 FRD_A2 = FRD_A2-2
enddo
do j=1,N
  A(j)=0
enddo
```

(c) na verdere optimalisatie

Figuur 5.4: Het programma in figuur 5.3 met dynamische doelhint-berekening voor referentie A(i).

if-testen moeten ingevoerd worden. De dominante domeinen bevatten de overgrote meerderheid van de toegangen, en voor de meeste toegangen wordt dus de juiste hergebruiksafstand berekend. Indien toch meer precisie gewenst is, kunnen de iteratiepunten met een verschillende Ehrhart-polynoom afgesplitst worden, bijv. door de eerste of laatste iteratie uit de lus te halen (loop peeling). Een voorbeeld hiervan is te zien in figuur 5.4.

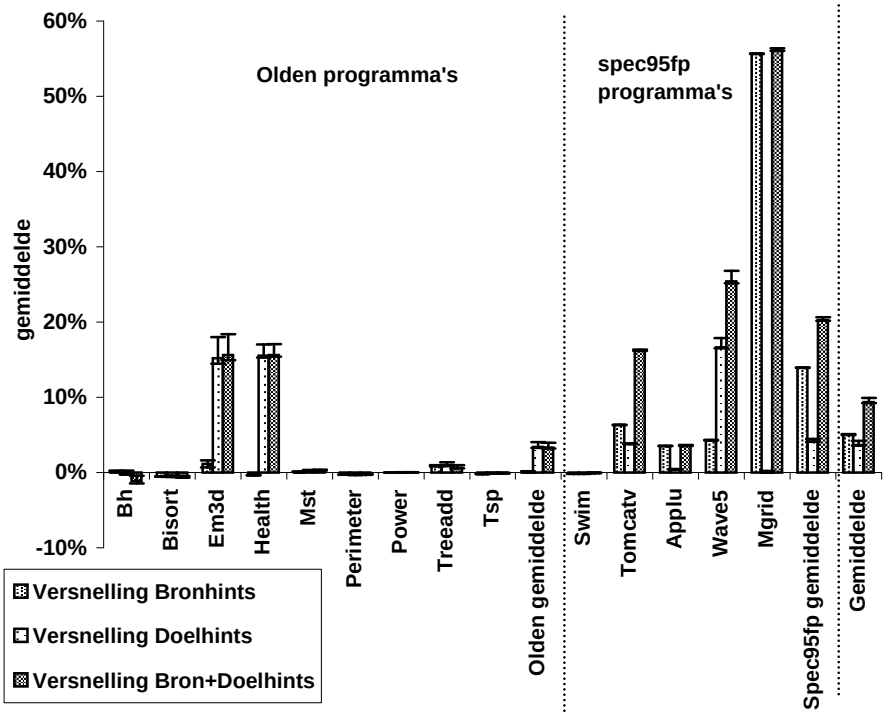
De overhead die hierna nog overblijft wordt veroorzaakt door het evalueren van de polynoom. Deze overhead kan meestal sterk verkleind worden door compileroptimalisaties zoals strength reduction en loop invariant code motion.

5.4 Experimenten

Het opmeten van hergebruiksafstandsdistributies, en de bijhorende statische cachehint-selectie werd geïmplementeerd in de ORC-compiler [132]. De versnelling na het introduceren van zowel bron- als doelhints, op een Itanium1 multiprocessor, wordt getoond in figuur 5.5. De gemiddelde versnelling over 100 programma-runs werd berekend, evenals het 99% betrouwbaarheidsinterval van deze versnelling. Uit de figuur blijkt dat de programma's gemiddeld 9% versneld werden, met een maximale versnelling van 56%. De Olden-benchmarks, die vooral uit pointer-gebaseerde code bestaat die dynamische datastructuren doorloopt, versnellen gemiddeld gezien enkel door de doelhints. Voor deze programma's is er te weinig parallellisme aanwezig opdat de instructiescheduler nuttig gebruik kan maken van de informatie in de bronhints. In tegenstelling tot de Olden-benchmarks, bezitten de SPEC95FP-programma's veel instructie-niveau-parallelisme, waardoor de bronhints wel aanleiding geven tot versnelling.

De cachemisser-reductie na het introduceren van zowel statische als dynamische doelhints wordt getoond in tabel 5.1. Uit die tabel blijkt dat het dynamisch selecteren van doelhints ongeveer dubbel zoveel cachemissers kan verwijderen als het statisch selecteren van doelhints (10.34% i.p.v. 5.14%).

De overhead van de dynamische doelhints wordt getoond in tabel 5.2. Wanneer voor iedere individuele toegang de exacte hergebruiksafstand berekend wordt, leidt dit tot een grote overhead (22 maal grotere code, en 63 maal tragere uitvoering). Door enkel de dominante domeinen te beschouwen, is het echter mogelijk om de code-overhead



Figuur 5.5: Versnelling met statische cache hints. De 99% betrouwbaarheidsintervallen worden aangegeven door streepjes.

	LRU	LRU+statische hints		LRU+dynamische hints	
programma	miss rate	miss rate	reductie	miss rate	reductie
vpenta	31.56%	25.57%	*19.00%*	25.57%	*18.99%*
mxm	3.20%	3.17%	*0.87%*	3.20%	*0.00%*
liv18	68.46%	63.13%	*7.78%*	61.91%	*9.57%*
cholesky	19.81%	25.48%	*-28.59%*	17.94%	*9.43%*
jacobi	14.32%	14.32%	*0.02%*	14.32%	*0.00%*
gauss-jordan	11.90%	8.94%	*24.82%*	7.81%	*34.37%*
tomcatv	9.22%	8.11%	*12.05%*	9.22%	*0.00%*
gemiddeld	22.64%	21.25%	***5.14%***	20.00%	***10.34%***

Tabel 5.1: De cache miss-rates voor een 4-weg-set-associatieve 16KB cache met 32 bytes per lijn. De eerste kolom toont de programmanaam; de tweede kolom toont de missrate met LRU vervanging; de derde en vierde kolom bevatten de absolute miss rate en de relative miss rate reductie ten opzicht van LRU voor statische doelhints; de vierde en de vijfde kolom tonen de missrate en missrate-reductie met dynamische doelhints.

programma	exact		dominante domeinen		
	rel. code-grootte	rel. tijd	rel. code-grootte	rel. tijd	rel. IPC
vpenta	4.81	2.48	1.11	1.02	0.98
mxm	1.83	34.45	1.01	1.00	1.00
liv18	47.10	55.36	1.15	1.02	1.16
cholesky	2.17	5.77	1.34	0.98	1.22
jacobi	2.65	10.69	1.01	0.99	0.89
gauss-jordan	2.71	72.62	1.15	1.01	1.01
tomcatv	92.72	260.88	1.39	1.00	1.08
gemiddeld	22.00	63.18	1.17	1.00	1.05

Tabel 5.2: De overhead van de dynamische berekening van voorwaartse hergebruiksafstanden en de overeenkomstige doelhints. De eerste twee kolommen tonen de overhead wanneer exacte hergebruiksafstanden worden berekend. De derde en vierde kolom tonen de overhead wanneer enkel de dominante domeinen worden gebruikt (zie sectie 5.3.2). De laatste kolom toont de relatieve IPC (instructions per cycle), van de dominante domeinen-versie t.o.v. het originele programma zonder cache hints.

programma	% foutieve rd	% foutieve hint
vpenta	0.31%	0.12%
mxm	0.50%	0.12%
liv18	0.21%	0.01%
cholesky	0.37%	<0.01%
jacobi	0.24%	0.03%
gauss-jordan	0.08%	<0.01%
tomcatv	2.52%	0.09%
gemiddeld	0.60%	0.05%

Tabel 5.3: Percentage verkeerd voorspelde voorwaartse hergebruiksafstanden en doelhints door enkel de dominante Ehrhart-polynomen te beschouwen.

te beperken tot 17%, terwijl de uitvoeringsoverhead verdwijnt. De extra berekeningen veroorzaakt door het evalueren van de dominante Ehrhart-polynoom worden parallel uitgevoerd met de berekeningen in het originele programma, op ongebruikte functionele eenheden. De evaluatie van de Ehrart-polynomen geeft aanleiding tot circa 5% extra uit te voeren instructies. Ondanks het feit dat enkel de dominante Ehrhart-polynoom gebruikt wordt om de hergebruiksafstand te berekenen van een instructie, wordt in 99.95% van alle toegangen de correcte doelhint berekend, zoals te zien is in tabel 5.3.

5.5 Samenvatting

In EPIC-architecturen bestaan er twee soorten cache hints: *bron-* en *doel-*hints. De bronhints duiden het verwachte cacheniveau aan waar data gevonden wordt, en de doelhints duiden het verwachte cacheniveau aan waarvoor het nuttig is om de data daar te bewaren. Er werd een verdere classificatie voorgesteld, als *statische* of *dynamische* hints. De statische hints geven het belangrijkste cacheniveau aan voor alle uitvoeringen van een instructie, terwijl de dynamische hints voor iedere individuele toegang het precieze cacheniveau bepalen. De statische hintgeneratie gebeurt op basis van de geprofileerde hergebruiksafstandsdistributies van een instructie. De dynamische hint-selectie gebeurt op basis van de hergebruiksafstandspolynomen berekend via de hergebruiksafstandsvergelijkingen.

De cachehintgeneratie werd geïmplementeerd in de ORC-compiler, en de experimenten duiden op een gemiddelde versnelling van 10% na het genereren van statische doel- en bronhints. Bovendien blijkt dat dynamische doelhints ongeveer dubbel zoveel cachemissers verwijderen als statische doelhints, zoals te zien is in tabel 5.1.

Hoofdstuk 6

Visualisatie van Lange-afstandshergebruik

Voor vele programma's kunnen de bestaande compileroptimalisaties het aantal cachemissers nauwelijks verminderen. Vooral de capaciteitsmissers, die een gevolg zijn van lange-afstandshergebruik, worden nauwelijks gereduceerd. Om dergelijke missers te verminderen dient het gebruik en hergebruik dichter bij elkaar gebracht te worden. Om dit te kunnen doen moet er eerst zicht zijn op het gebruik, het hergebruik, en de code die daartussen wordt uitgevoerd. Aangezien het hergebruik gebeurt op lange afstand, moet er een overzicht zijn over de miljoenen uitgevoerde instructies tussen gebruik en hergebruik. De compileranalyses zijn meestal niet in staat om zo'n groot overzicht over het programma te verkrijgen. Mocht de compiler wel in staat zijn tot een analyse die genoeg overzicht krijgt, dan is het nog steeds moelijk voor de compiler om een geschikte geldige transformatie van het programma te vinden dat de hergebruiken dichter bij elkaar brengt.

Daarom wordt in dit hoofdstuk de taak van het verkleinen van lange hergebruiksafstanden doorverwezen naar de programmeur. Die heeft immers meestal een goed overzicht over het globale programmaverloop. Bovendien heeft die meer vrijheid dan de compiler in het aanpassen van de algoritmen. De lange-afstandshergebruiken en het cachegedrag van een programma kan echter niet afgelezen worden uit de broncode. Daarom wordt hier een visualisatie voorgesteld die de programmeur de cachebottlenecks in zijn programma toont. In vergelijking met bestaande visualisaties van het cachegedrag is dit de eerste visualisatie die de programmeur een hint geeft over hoe de lokaliteit van zijn programma platform-onafhankelijk kan verbeterd worden.

6.1 Visualisatie van Hergebruik met Lage Lokaliteit

6.1.1 Platformonafhankelijke Cache-optimalisaties

Aangezien de meerderheid van alle missers capaciteitsmissers zijn, en het net deze soort missers zijn die door compilers nauwelijks weggewerkt kunnen worden, concentreren we ons hier op het wegwerken van die missers. Bovendien zijn capaciteitsmissers relatief onafhankelijk van de onderliggende architectuur. Enkel de cachegrootte heeft een grote invloed op het aantal cachemissers. Aangezien de cachegrootte in de meeste general purpose systemen in dezelfde grootteorde liggen (256KB – 4MB), zullen de meeste van die systemen dezelfde capaciteitsmissers ondervinden. Door deze missers aan te pakken zal het cachegedrag dus voor een brede waaier aan architecturen verbeterd worden. Capaciteitsmissers (of hun vertragend effect) kunnen verwijderd worden op 4 verschillende manieren:

1. De eerste manier is om de *geheugentoegangen met lage lokaliteit te verwijderen*. Dit is soms mogelijk door een andere datastructuur of een ander algoritme te kiezen. Een voorbeeld van zo'n optimalisatie is de volgende. Beschouw een 2-dimensionale array in C, die geconstrueerd werd als een array van arrays: `double** A; A[i][j]=0;`. De code `A[i][j]` geeft aanleiding tot twee load instructies: een om `A[i]` op te halen, en een om de waarde `A[i][j]` te laden. Indien het ophalen van `A[i]` een capaciteitsmisser veroorzaakt, kan deze weggewerkt worden door de array te veranderen in een een-dimensionele array: `double* A; A[i*N+j]=;`.

2. De tweede manier is om de afstand tussen gebruik en hergebruik te verkleinen, zodat die kleiner wordt dan de cachegrootte. Dit kan meestal gedaan worden door middel van het herordenen van berekeningen, zodat de *temporele lokaliteit verhoogt*.

3. Een derde manier is om de *spatiale lokaliteit te verhogen*. Dit kan zowel gebeuren door data op een andere manier in het geheugen te leggen, zodat data die samen gebruikt wordt dicht bijeen liggen in het geheugen. Een andere manier om dit te bereiken is de volgorde van de berekeningen te veranderen bij een vaste datalayout.

4. Indien geen van de drie bovenstaande mogelijkheden toegepast kunnen worden, kan het nog steeds mogelijk zijn om de reken-snelheid te verhogen door *de latentie van de capaciteitmissers te verbergen*. De bekendste techniek in deze klasse is prefetchen.

6.1.2 Lokaliteitsmetrieken

De lange hergebruiksafstanden duiden lage temporele lokaliteit aan. Naast temporele lokaliteit, kunnen caches ook nuttig gebruik maken van spatiale lokaliteit. Die spatiale lokaliteit wordt gemeten aan de hand van de volgende metriek:

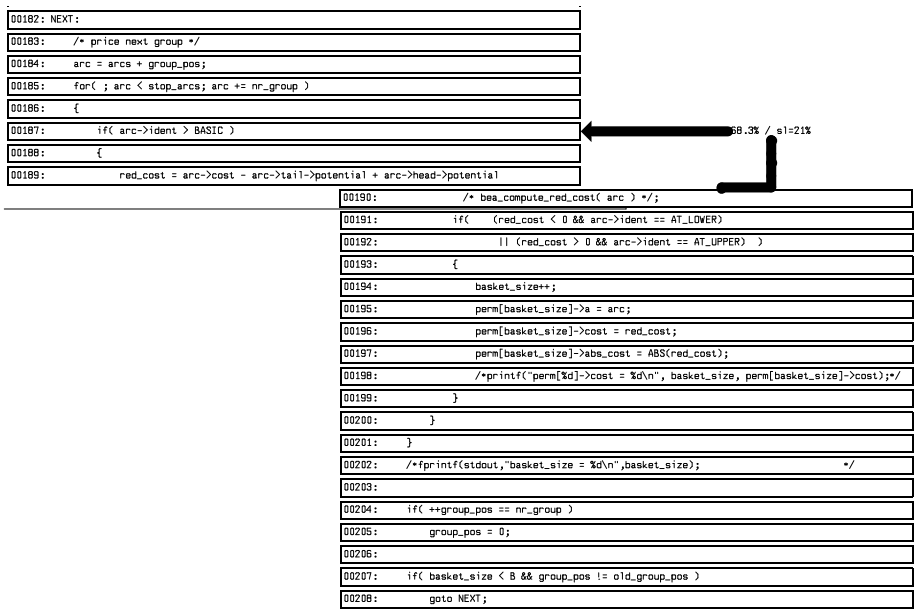
Definitie 14. *Het* **nuttig geheugenlijngebruik** *van een toegang t naar geheugenlijn l is de fractie van l die gebruikt wordt voordat l uit de cache geworpen zal worden.*

Wanneer de achterwaartse hergebruiksafstand van een toegang t groter is dan de cachegrootte treedt een misser op. Bijgevolg zal t geheugenlijn l in de cache brengen. Indien het nuttig geheugenlijngebruik lager is dan 100%, worden niet alle bytes in l gebruikt gedurende het verblijf van l in de cache. Dus werden er tijdens toegang t een aantal nutteloze bytes in de cache gebracht, en het potentiele nut van het ophalen van een volledige geheugenlijn werd niet ten volle benut. Het geheugenlijngebruik duidt aan hoeveel de spatiale lokaliteit verbeterd kan worden.

6.1.3 Visualisatie

Het opmeten van het nuttig geheugenlijngebruik en de hergebruiksafstanden werd geïmplementeerd door programma's te instrumenteren met behulp van de ORC-compiler. Voor ieder paar instructies (i_1, i_2) wordt de distributie van hergebruiksafstanden $\mathsf{RDD}(i_1, i_2)$ opgemeten. Het gemiddelde geheugenlijngebruik voor de hergebruiksparen werd eveneens opgemeten.

Enkel de hergebruiksparen met een afstand groter dan de cachegrootte geven aanleiding tot capaciteitsmissers. Daarom worden enkel die hergebruiksparen getoond die een grote hergebruiksafstand hebben. Bovendien worden enkel de instructie-paren getoond die minstens 1% van alle lange-afstandshergebruiken veroorzaken, opdat de programmeur onmiddellijk de belangrijkste cachebottlenecks kan zien. De

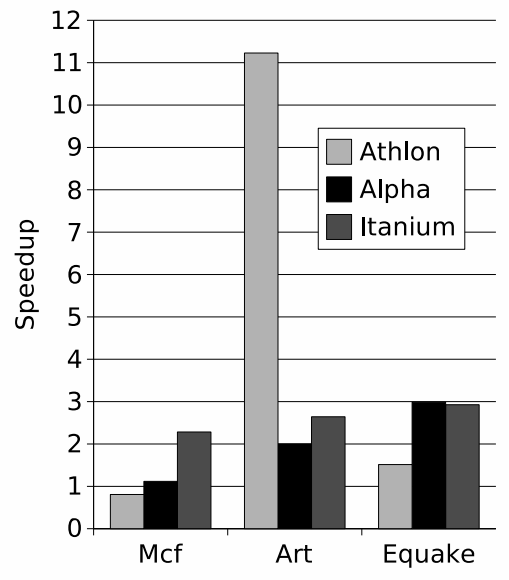


Figuur 6.1: Visualisatie van de belangrijkste lange-afstandshergebruiken in mcf. Er is een enkel paar instructies die de meeste lange-afstandshergebruiken veroorzaakt (68.3% van alle capaciteitsmissers wordt door deze pijl veroorzaakt). Het tweede getal naast de pijl (sl=21%) toont aan dat de cachemissende instructie (op lijn 187) een geheugenlijngebruik heeft van 21%.

hergebruiksparen worden voorgesteld als pijlen die bovenop de broncode worden getekend, gaande van de instructie die de data gebruikt, naar de instructie die de data hergebruikt. Bij iedere pijl hoort een label dat het percentage van lange-afstandshergebruiken aantoont dat door dat paar instructies veroorzaakt wordt. Een tweede percentage in het label toont het nuttig geheugenlijngebruik. Een voorbeeld is te zien in figuur 6.1.

6.2 Experimenten

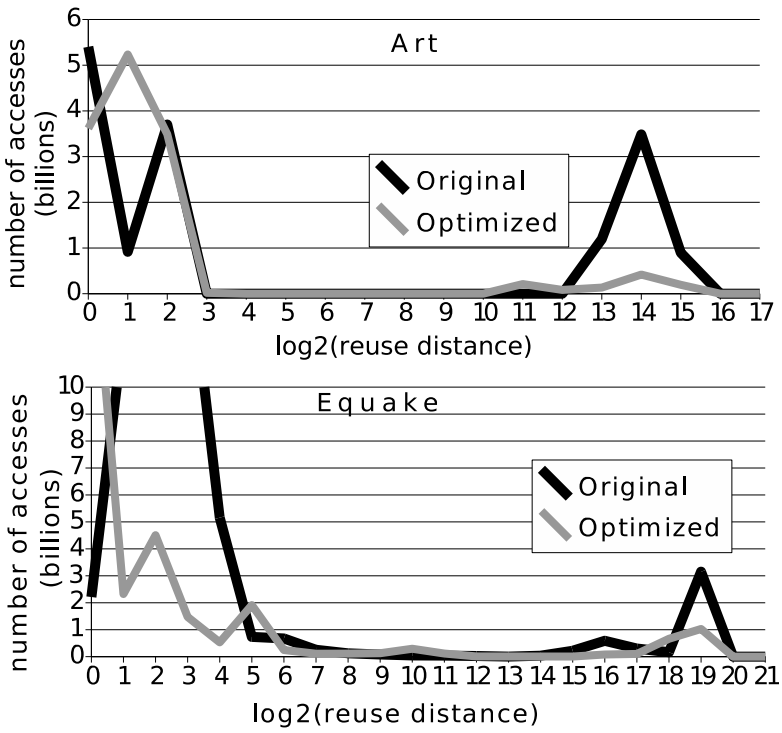
De visualisatie werd toegepast op de drie programma's uit de SPEC2000 benchmark met de grootste cachebottlenecks: mcf, art en equake. Gebaseerd op de visualisatie werden er een aantal relatief kleine veranderingen aan de broncode aangebracht, met als doel het verwijderen van capaciteitsmissers. De twee versies van de broncode (originele en geoptimaliseerde) werden gecompileerd op 3 verschillende platfor-



Figuur 6.2: Versnelling op verschillende architecturen.

men: een Athlon PC, een Alpha werkstation en een Itanium server. Telkens werd er gecompileerd met een state-of-the-art compiler, op het hoogste optimalisatie-niveau. De versnelling van de drie programma's op de verschillende platformen is te zien in figuur 6.2.

De figuur toont dat er voor bijna alle programma's op alle platformen een duidelijke versnelling optreedt, met een gemiddelde versnelling van 3.06. Voor het programma mcf werden extra prefetch-instructies toegevoegd, en de vertraging op de Athlon voor dit programma is waarschijnlijk te wijten aan interacties tussen deze prefetch-instructies en de hardwarematige prefetcher die aanwezig is in de Athlon processor. Het hergebruiksafstandshistogram van de programma's art en equake na optimalisatie is te zien in figuur 6.3. De figuur toont duidelijk dat het aantal lange-afstandshergebruiken drastisch gedaald is, wat aanleiding geeft tot een grote reductie van het aantal capaciteitsmissers.



Figuur 6.3: Hergebruiksafstanden voor en na optimalisatie.

6.3 Samenvatting

Om capaciteitsmissers te verwijderen moeten gebruik en hergebruik dichter bij elkaar gebracht worden. Deze hergebruiken gebeuren ver van elkaar, en een groot overzicht over de programma-uitvoering is nodig om de hergebruiken dichter bij elkaar te kunnen brengen. Zoals in hoofdstuk 3 werd aangetoond, slagen compilers er meestal niet in om die lange-afstandshergebruiken te verkleinen. Daarom wordt deze taak beter naar de programmeur gedelegeerd, die meestal een beter overzicht heeft over de grote fases in de programma-uitvoering. Om de programmeur te helpen werd in dit hoofdstuk een visualisatie voorgesteld die de lange-afstandshergebruiken zichtbaar maken in zijn code. Deze visualisatie werd toegepast op de drie programma's met de grootste cacheproblemen uit de SPEC2000 benchmark. Gebaseerd op de visualisatie werden een klein aantal veranderingen aan de broncode aangebracht. Deze veranderingen leidden tot een gemiddelde versnelling met een factor 3 op verschillende processorplatformen.

Hoofdstuk 7

Besluit

De snelheidskloof tussen processor en hoofdgeheugen groeit met exponentiele snelheid. Caches trachten het merendeel van de geheugentoegangen snel te maken, maar tijdens een cachemisser moet het trage hoofdgeheugen benaderd worden. Voor de meeste programma's blijken de cachemissers veroorzaakt te worden door een beperkte cachecapaciteit. Deze capaciteitsproblemen kunnen enkel opgelost worden door verbeteringen aan te brengen in de software. De belangrijkste bijdragen van dit onderzoek naar softwaretechnieken voor het verbeteren van cachegedrag werd gedaan in de volgende gebieden:

Cache Remapping Cache remapping is een methode van softwareoptimalisatie die tracht de processorwachttijd veroorzaakt door cachemissers tot nul te herleiden. Dit doel wordt nagestreefd door ideeën te combineren uit vier categorieën van cacheoptimalisaties: tegeling om het aantal capaciteitsmissers te verminderen, herlayout van data in de cache om conflictmissers te verwijderen, cache hints om de cachevervanging te sturen en prefetchen in een aparte draad om de latentie van de overblijvende missers te verbergen. De aparte draad kan at compile-time verweven worden met de originele rekendraad, zodat deze techniek ook toepasbaar is op processors zonder meerdradige uitvoeringsmogelijkheden.

Hergebruiksafstand Als een arts zijn patient wil genezen, moet hij eerst de oorzaak van zijn ziekte vinden. Om capaciteitsmissers te verminderen, moet ook hun oorzaak gezocht worden. In hoofdstuk 3 werd de hergebruiksafstand geïntroduceerd als metriek voor het karakteriseren van capaciteitsmissers. In dat hoofdstuk

werd ook aangetoond dat een behoorlijk aantal van de conflictmissers op automatische wijze kan verwijderd worden door de compiler, in tegenstelling tot de beperkte afname van het aantal capaciteitsmissers. Naarmate de hergebruiksafstanden langer worden, is het voor de compiler moeilijker om genoeg grip te hebben op het programma om hun cachegedrag te kunnen optimaliseren.

Hergebruiksafstandsvergelijkingen In hoofdstuk 4 werden vergelijkingen voorgesteld die de hergebruiksafstand van alle geheugentoegangen modelleert aan de hand van polytopen. Het oplossen van deze vergelijkingen vergt o.a. een nieuwe methode voor het berekenen van het aantal oplossingen in een geparameteriseerd stelsel lineaire ongelijkheden. Het resultaat van de vergelijkingen geeft een compacte voorstelling, door middel van een verzameling van Ehrhart-polynomen, van het cachegedrag voor iedere individuele toegang.

Cachehint-Generatie Gebaseerd op de hergebruiksafstanden werd een cachehint-generatie voorgesteld in hoofdstuk 5. De bronhints geven aanleiding tot een verbeterde instructiescheduling, terwijl de doelhints een verbeterd vervangingsbeleid tot gevolg hebben. Uit de experimenten blijkt dat de statische hints, gebaseerd op hergebruiksafstandsdistributies, geven een versnelling van 9% op een Itanium processor. De statische doelhints geven aanleiding tot 5% minder cachemissers. Door gebruik te maken van de analytisch berekende hergebruiksafstanden is het mogelijk om op efficiënte manier dynamische cache hints te genereren. Deze dynamische hints reduceren het aantal cachemissers met 10%.

Visualisatie In hoofdstuk 6 wordt het herschikken van de berekeningen op hoog niveau, met als doel het verkorten van de langeafstandshergebruiken, overgelaten aan de programmeur. Om hem hierin te ondersteunen, werd er een visualisatie van de hergebruiken op lange afstand voorgesteld. De visualisatie toont de programmeur de plaatsen in zijn broncode waar hij temporele en spatiale lokaliteit kan verbeteren. Door gebruik te maken van de hergebruiksafstand is deze visualisatie de eerste die expliciet tracht de programmeur zich te laten richten op platform-onafhankelijke cache-optimalisaties. Deze visualisatie werd toegepast op 3 programma's uit SPEC2000, waarna er

een aantal relatief kleine broncode-wijzigingen werden aangebracht. Deze geoptimaliseerde broncode geeft een gemiddelde versnelling van 3 op een verscheidenheid van processors en cachehiërarchieën.

Part II

Software Methods to Improve Data Locality and Cache Behavior

Chapter 1

Introduction

Any computer system consists of at least two conceptual parts: a processing unit and memory. Any programmer wishes that the computer system at his disposal processes instructions fast and that the memory is large, so that it is able to solve many problems. However, fast memories are expensive, and ultimately large memories cannot be made as fast as small memories. Therefore memory hierarchies are present in almost all computer systems. In order to make them effective, and to keep the processor from being data-starved, the majority of the memory accesses must be serviced by the smallest levels in the hierarchy.

Processor execution speed increases at a rate of about 55% per year, while main memory access speed increases at a rate of only about 7% per year [83]. As a result, the speed gap between processor and memory grows at an exponential rate of about 45% per year. Therefore, ever more powerful optimizations are needed to ensure that most data can be found in the fastest levels of the hierarchy.

1.1 Background

Since the dawn of electronic computers, memory access speed has limited the processing speed [150]. Memory hierarchies were introduced in the early 1950s [150]. For example, the CEC-201 computer [149] has a high-speed core memory with an access time of 0.85 milliseconds containing 80 words, and a main drum memory containing 4000 words with average access time of 8.5 milliseconds. The inclusion of the high-speed memory allowed computations to proceed 7 to 8 times faster.

However, for these machines, the programs needed to explicitly transfer data between memory levels. In most modern computers, caches automatically place data at an appropriate memory level, and programs need not be aware of the hierarchy. The automatic management of the memory hierarchy was first proposed in 1962 by Kilburn et al. [100], where a paging scheme was introduced which automatically moves data between drum and core memory, implemented in the Atlas computer. In 1965, Wilkes [188] introduced the concept of a direct mapped cache memory, which was first implemented in the Atlas 2 and ETL Mk-6 computers. The first commercial computer to implement it was the IBM/360 model 85, as described by Liptay in [112], where the term "cache" is used for the first time. Of course, next to the cache hierachy, also the registers, the main memory, the hard drive and the network can be considered as being part of the memory hierarchy. In contrast to caches, the contents of these hierarchy levels are typically controlled by software.

Since the effective use of cache memory is so important to overall computer performance, thousands of research papers have discussed cache optimizations in the last forty years [12]. The proposed optimizations range from hardware modifications, over microarchitectural enhancements, optimizations in compilers and operating systems, to improvements at the algorithmic level. Notwithstanding the effort of many researchers to improve cache performance, on current systems, cache misses are a major bottleneck. As an illustration, figure 1.1 shows the cause of performance loss in the SPEC2000 programs as executed on an Itanium-1 processor. The programs were compiled with the highest level of feedback-driven optimization with Intel's state-of-the-art compiler. The processor has a 3-level cache hierarchy and 85% of its transistors are dedicated to these caches. In spite of state-of-the art optimizations at the hardware, the microarchitectural and the compiler level, the processor is stalled on data memory accesses for almost 50% of the execution time. In comparison, only 5% of the execution time is lost due to instruction cache misses.

The main reason why caches improve memory access, is that data is typically reused many times during program execution. Cache misses occur when the cache cannot exploit the locality of memory accesses. Cache misses are most often classified according to the their root cause, using the 3C's model: a miss is either a *cold* miss, a *conflict* miss or a *capacity* miss. A cold miss occurs when the data is accessed for the first time. Capacity misses arise when too much data has been accessed be-

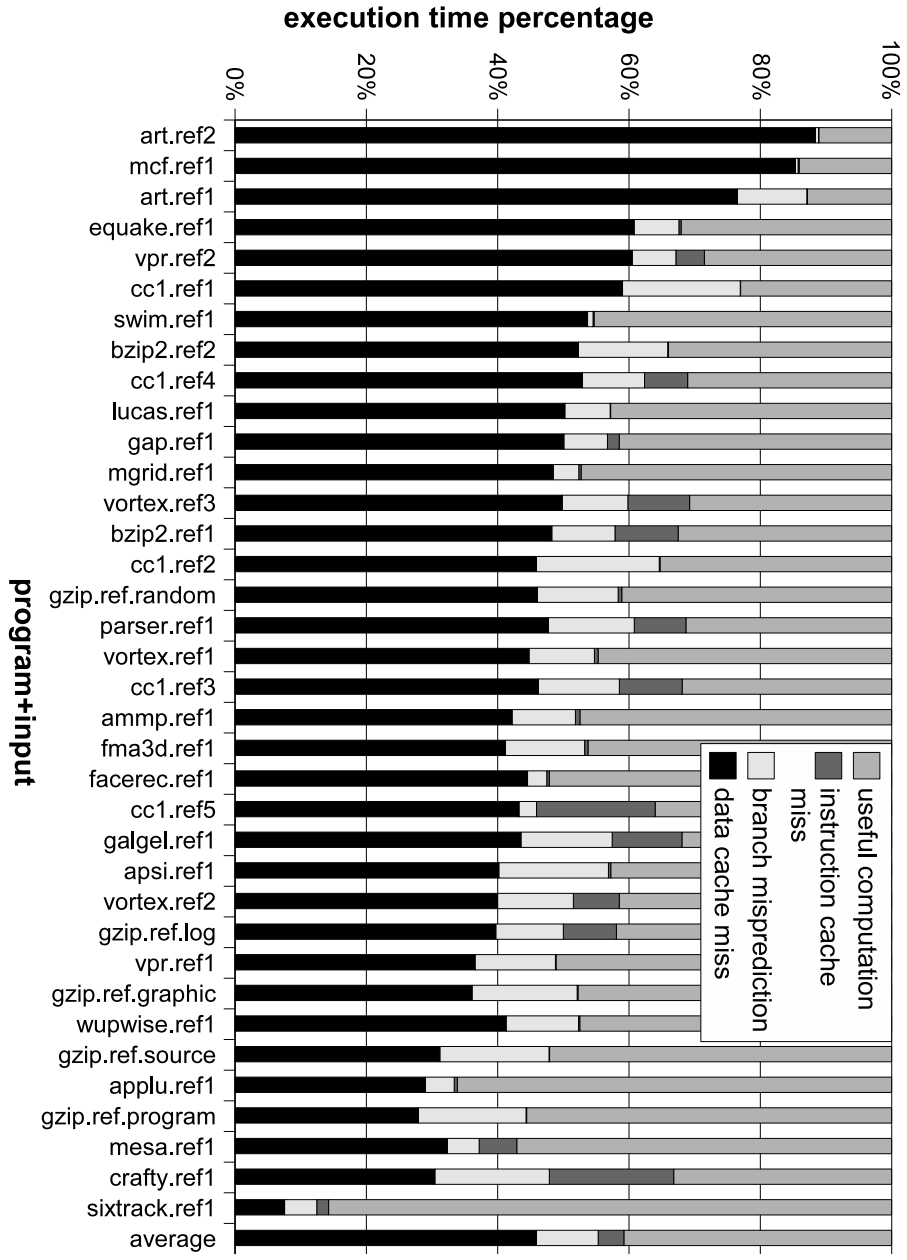


Figure 1.1: Causes of performance loss on an Itanium1 multiprocessor for the SPEC2000 benchmark, as measured using the performance counters [25].

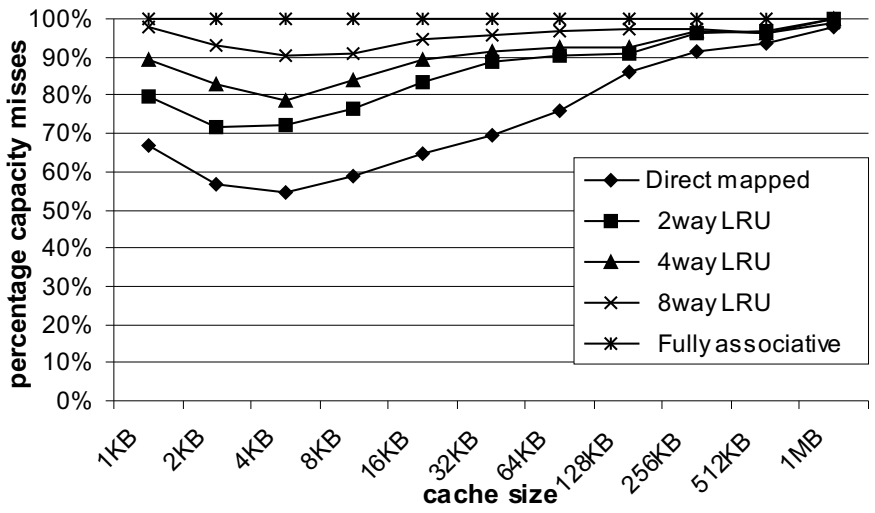


Figure 1.2: The percentage of data cache misses caused by limited capacity for varying cache size and associativity, for the SPEC2000 benchmark, as measured by Cantin and Hill [37].

tween use and subsequent reuse, and the cache is too small to keep all intermediate data. In direct mapped and set-associative caches, a given line of data can only be mapped to a specific subset of the cache, based on the memory address of the data. In these caches, conflict misses arise when use and reuse occurs close together, but the mapping constraints resulted in premature eviction of the reused data. In most applications, cold misses form only a very small fraction, and the dominating type of misses are the conflict and capacity misses. Figure 1.2 shows that for caches typically found in current microprocessors (2-way set-associative or more, larger than 8 KB), at least 80% of all misses are capacity misses (for the SPEC2000 programs).

At the hardware and microarchitectural level, there's little to be done about capacity misses, except to increase the cache size [83]. However, enlarging the cache makes it slower. Therefore, the dominating capacity misses should be removed by optimizations at the software level.

1.2 Overview and Contributions

In this dissertation, software techniques are investigated for optimizing the cache behavior of data accesses. Since capacity misses are the dominating misses and they should be resolved by software optimizations, the main focus is on characterizing cache capacity bottlenecks and how to improve cache behavior based on that characterization. Most computer systems include several types of cache hardware, such as multi-level data and instruction caches, TLBs, trace caches, victim caches, stream buffers, and so on. In this dissertation, the main focus is on characterizing and optimizing the behavior of data caches.

Chapter 2 starts with an overview of the best-known software methods for optimizing cache behavior, and categorizes them into 4 classes: (1) optimizations that reduce capacity misses; (2) optimizations to reduce conflict misses; (3) optimizations which hide misses by performing parallel computations and (4) optimizations that improve the cache replacement policy. After that short survey, *cache remapping* is introduced, which is a method that optimizes the cache behavior of tiled loops and combines ideas from all four categories. It compares favorably with other methods that aim at reducing conflict misses in tiled loop kernels. Cache remapping is published in the following works:

[20] K. Beyls and E. D'Hollander. **Cache remapping to improve the performance of tiled algorithms.** Proceedings of the 6-th International Euro-Par Conference, pages 177–186, 2000

[21] K. Beyls and E. D'Hollander. **Compiler generated multithreading to alleviate memory latency.** Journal of Universal Computer Science, 6(10):968–993, oct 2000.

In chapter 3 the *reuse distance* is introduced as a metric for cache behavior. The reuse distance is shown to exactly model the cache behavior of fully associative caches. Also for lower-associative caches, the reuse distance models the cache behavior fairly accurately. Furthermore, the influence of state-of-the-art compiler optimizations on the reuse distance of memory accesses in the Spec95fp programs is investigated, and it appears that current compiler optimizations can hardly optimize accesses with large reuse distances. Furthermore, the compiler optimizations can eliminate about 30% of the conflict misses, while they only remove about 1% of the capacity misses. The limitations of existing

compiler optimizations in reducing capacity misses has been presented in

[28] K. Beyls and E. H. D'Hollander. **Reuse distance as a metric for cache behavior.** In International conference on Parallel and Distributed Computing and Systems, pages 617–662, Aug 2001.

In the remaining chapters, the focus is on characterizing and eliminating capacity misses. Our research is focused mostly on the capacity misses since: (a) they are the dominant type of misses for most applications; (b) capacity misses should be resolved by software, whereas conflict misses can be resolved by micro-architectural optimizations; (c) current state-of-the-art compiler optimizations remove a substantial amount of conflict misses, but hardly any capacity misses are eliminated.

Next to profiling reuse distances, as discussed in chapter 3, chapter 4 shows how to analytically calculate reuse distances for programs in the polyhedral model. First, the required polyhedral theory is introduced, after which the reuse distance is modelled using polytopes and Presburger formula. The current polyhedral tools are limited in solving these reuse distance equations, and the limitations are investigated in more detail. One of the corner stones in solving reuse distances is computing the number of integer points in a parameterized polytope. A previous method to compute this and its limitations are discussed, after which a new method is presented which overcomes the limitations of the previous method. The improved method has been developed in close collaboration with Sven Verdoolaege and prof. Maurice Bruynooghe from Katholieke Universiteit Leuven and Rachid Seghir and dr. Vincent Loechner from Université Louis Pasteur Strasbourg, and resulted in the following publications:

[184] S. Verdoolaege, K. Beyls, M. Bruynooghe, R. Seghir, and V. Loechner. **Analytical computation of Ehrhart polynomials and its applications for embedded systems.** 2nd Workshop on Optimizations for DSP and Embedded Systems, Palo Alto, March 2004.

[158] R. Seghir, S. Verdoolaege, K. Beyls, and V. Loechner. **Analytical computation of Ehrhart polynomials and its applications in compile-time generated cache hints.** Technical report, Université Louis Pasteur Strasbourg, 2004.

Chapter 3 and chapter 4 discuss methods to measure or calculate the reuse distance. In chapters 5 and 6, two different approaches are presented to optimize the cache behavior based on the reuse distance metric.

In chapter 5, the selection of cache hints based on the reuse distance metric is discussed. Cache hints are annotations to memory instructions that occur in two kinds: *source hints* and *target hints*. The source hints indicate the fastest cache level where data is expected to be found, while target hints indicate the cache level where the data is expected to be retained until its next use. The source hints are used in the instruction scheduler to better estimate the latency of load instructions. A traditional instruction scheduler in the compiler assumes that all load instructions have a cache hit latency. Using source hints, the scheduler is better informed and it constructs an improved schedule. The target hints ameliorate the replacement policy, by only allocating data in the cache levels where it is expected to be retained. This reduces cache pollution in upper cache levels and reduces the number of cache misses.

Two alternative methods are proposed to generate cache hints. The first is based on profiled reuse distance distributions and generates *static hints*, i.e. a fixed hint per memory instruction. A second method, based on the reuse distance equations generates *dynamic hints*, i.e. the most appropriate hint for each dynamic execution of a memory instruction is calculated at run-time. In order to minimize the run-time overhead, optimizations which exploit the structure of the analytically calculated reuse distances are proposed. Finally, the cache hint generation has been implemented in the ORC-compiler for Itanium, and has been evaluated on both pointer-based and numerical programs, leading to 5% faster execution on average. The cache hint selection scheme based on the reuse distance metric has been introduced in the following publications:

[23] K. Beyls and E. D'Hollander. **Reuse distance-based cache hint selection.** Proceedings of the 8th International Euro-Par Conference, pages 265–274, 2002.

[22] K. Beyls and E. D'Hollander. **Compile-time cache hint generation for EPIC architectures.**, Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2), 2002. (best paper award)

In chapter 6, it is acknowledged that automatic compiler optimiza-

tions are limited in reducing the number of long reuse distances, since this requires a thorough understanding of long term program behavior. Therefore, it might be better to delegate capacity miss optimizations (=reducing long reuse distances) to the programmer, who has a better understanding of the long term behavior of his program, and has more freedom than the compiler to change algorithms. However, manually optimizing programs requires costly human resources. Therefore, a visualization of the long reuse distances is proposed. The visualization increases the human efficiency in two ways. First, cache behavior is not obvious from the source code. The visualization shows the references in the code that generate long reuse distances. Secondly, the reuse distance is a platform-independent locality metric, which steers the programmer into optimizing the locality in a portable way. The visualization has been applied to three programs from SPEC2000, and the effectiveness of the platform-independent source code optimizations are indicated by an average speedup of more than 3 on computers ranging from PCs over workstations to servers, equipped with Athlon, Alpha and Itanium processors, each with a different memory hierarchy. The visualization of cache behavior has been studied in collaboration with dr. Yijun Yu and resulted in the following publications:

[198] Y. Yu, K. Beyls, and E. D'Hollander. **Visualizing the impact of cache on the program execution.** In *Information Visualization 2001*, pages 336–341, 2001.

[27] K. Beyls, E. D'Hollander, and Y. Yu. **Visualization enables the programmer to reduce cache misses.** International Conference on Parallel and Distributed Computing and Systems, 2002. (best paper award)

[25] K. Beyls and E. D'Hollander. **Platform-independent cache optimization by pinpointing low-locality reuse.** Proceedings of the International Conference on Computational Science, pages 448–455, June 2004.

In chapter 7, the main conclusions and contributions of the previous chapters are summarized, and interesting directions for future research are discussed.

Chapter 2

Cache Remapping

In the past two decades, many software optimizations have been proposed to improve average memory access time. Most of these optimizations can be classified as either being targeted at reducing conflict misses, reducing capacity misses, or hiding the memory latency by some form of prefetching.

In the first section, four categories of program optimizations are surveyed. In the following sections, ideas from all four categories are combined into the cache remapping *method. Capacity misses are reduced by tiling, conflict misses are eliminated by relayouting data at run-time, and the left-over misses are hidden by prefetching. The relayouting also allows to indirectly control which data is retained and which data is replaced in the cache. Prefetching and relayouting is performed in a separate* remap thread*, while the original computations are performed by the* processing thread*. For tiles of fixed size, the amount of work performed by both threads for a single tile is fixed, which allows to statically interweave the threads into a single thread, so that the program can be executed on a single-threaded processor. The execution speed of a tiled matrix multiplication kernel is compared with five other optimizations for tiled matrix computations. By removing all conflict misses, and prefetching the other misses, cache remapping generates the fastest execution.*

2.1 Overview of Existing Program Optimizations

During the last two decades, many program optimizations have been proposed to improve cache performance. Most of these optimizations can be classified into one of the following categories: (1) reducing capacity misses by increasing temporal or spatial locality; (2) reducing

conflict misses by improving data layout or changing computation order; (3) hiding memory latency by performing parallel computations; or (4) improving cache replacement policy.

2.1.1 Reducing Capacity Misses

Most optimizations that minimize capacity misses share the following strategy. Assume that on each data element, a fixed number of computations needs to be performed. The computations are interleaved with computations on other data elements. As a result, the data element might be evicted from the cache between two computations on it, and it needs to be refetched from memory. By performing as much useful computations as possible on a data element during a single stay in the cache, the data needs to be refetched less from the next level. Assume that for each data element in the program, N computations need to be performed, and the original program does 1 useful computation on each data element during a stay in the cache. For each data element, there are N misses. If the computation order is changed so that 2 useful computations are performed during a single stay in the cache, the number of misses is halved to $\frac{N}{2}$.

Viewed in another way, the distance in time between the computations on the same data is reduced. As a result, the data is more likely to remain in the cache between consecutive computations.

This effect can be achieved by either increasing temporal locality (more useful computations on the same data), or increasing spatial locality (more useful computations on the same cache line). The best-known optimizations that increase locality are:

- Loop transformations [120] such as loop tiling [32, 88, 98, 103, 110, 164, 191, 194] and loop fusion [55, 99, 109, 117, 120, 122, 129, 151, 185] can increase temporal locality. Loop tiling increases temporal locality in a single nested loop by minimizing the distance between reuses in outer loop iterations (an example is given in figures 2.1, 2.2 and 2.3). In contrast, loop fusion transforms two consecutive loop nests into a single nest, which reduces the distance between a use in the first loop and a subsequent reuse in the second loop.

- Spatial locality can be improved by reordering computations, so that computations on nearby data occur close together in time

[120]. Alternatively, the data layout can be changed so that data accessed by nearby computations are placed in nearby memory locations. The best-known spatial locality optimizations for arrays are transposition [8, 92], non-linear array layouts [43, 114], and array regrouping [61]. Spatial locality in "structures" (e.g. in C) is optimized by reordering fields in a single structure [45] or by packing the hot fields of several nearby accessed structures on the same cache line [46].

2.1.2 Reducing Conflict Misses

A second kind of miss is caused by limited cache associativity. When the data in the working set maps to a small number of cache sets, fewer bytes of the working set can be retained in the cache, and an excessive number of conflict misses might result. However, these misses can be reduced by making sure that the data in the working set is spread over all available cache sets. A number of hardware methods have been described to reduce conflict misses, such as skewed associative caches [160], victim caches [91] and XOR-based hashing functions [180]. Conflict misses can also be reduced by software optimizations, either by changing the data layout of the working set or by changing the working set itself (through loop transformations):

- The best-known data layout transformation to reduce conflict misses is array padding [10, 146, 147], which increases the array dimensions to make sure that array elements in the same column or row are more evenly distributed over all cache sets. In the context of tiled loop kernels, a number of methods have been proposed to ensure that the data tiles are spread out over all cache sets. Examples are copying data tiles into a contiguous buffer at run-time [171], or using hierarchical array layouts [43].

- In the context of tiled algorithms, a number of algorithms have been proposed to change the tile size. As a result the working set consists of data tiles that do not incur self-conflicts [52, 106] (i.e. the array elements in a given tile do not overflow any cache set).

2.1.3 Parallel Computation to Hide Memory Latency

When the cache miss cannot be eliminated, its slow-down effect can be hidden by performing useful computations in parallel with the memory fetch. Several techniques share this idea:

prefetching When it is known in advance which data is needed in the near future, it can be requested from main memory and placed in the cache in a timely fashion. Most software prefetching methods that are implemented in industrial compilers are based on an analysis of array traversals [36, 127]. Recently, methods to prefetch more irregular data structures have also been proposed [47, 116].

multithreading A second way to perform useful computations during a memory fetch is to quickly switch to another thread of execution. Many multithreaded processors [176] can switch threads on every clock cycle. Simultaneous multithreading processors are out-of-order processors which execute instructions when they are ready, independent of their originating thread [64]. The Tera MTA [7, 38] takes multi-threading for hiding memory latency to an extreme: the machine doesn't have caches. Instead, on every clock tick, it executes an instruction of the next thread in a pool, in a round-robin fashion. The thread pool contains up to 128 threads. In order to hide the latency, as many parallel threads as possible should be created.

2.1.4 Improving Replacement Decisions

Finally, the number of cache misses can be reduced by improving the replacement policy. The 3C's-model is based on a fixed replacement policy: LRU. Therefore, the 3C's-model cannot be used to classify the misses that could be removed by improving the replacement policy. (An improved classification scheme, that classifies misses as being caused by limited associativity, limited capacity, cold misses and misses due to non-optimal replacement has been proposed in [168]). A number of different approaches have been proposed to improve the replacement policy:

- As a pure hardware optimization, predictors have been suggested that predict the locality of memory accesses. Based on the prediction, the replacement policy is adapted [90, 111].

- A number of optimizations have been proposed where the compiler annotates hint bits to the instructions. These bits are used to adapt the replacement policy [89, 128, 174].

- A number of proposals assume a split cache hierarchy: one hierarchy for data with temporal locality, and another for data with only spatial locality. These temporal-spatial cache designs [140] can be steered by compiler-generated hints [152] or by hardware predictors [170].

In the remainder of this chapter, cache remapping is introduced. Cache remapping combines ideas from the above categories. The aim is to reduce the capacity misses by tiling, removing conflict misses by relayouting the data at run-time; and hiding the left-over cold and capacity misses by performing parallel computations. By relayouting the data, the software indirectly controls which data gets replaced and which data is retained.

2.2 The Cache Remapping Method

2.2.1 Tiled Loop Nests

Figure 2.1 shows a loop nest and the corresponding tiled loop.

Definition 1. *Tiling [35, 191] transforms an n-deep loop nest into a $2n$-deep loop nest. The* **tiled** *loops in the resulting tiled loop nest are the n inner loops. The* **tiling** *loops are the n outermost loops. A loop nest will be denoted as $\mathcal{L}$. The tiled and the tiling loops for $\mathcal{L}$ are denoted by $Td(\mathcal{L})$ and $Ti(\mathcal{L})$ respectively. An* **iteration tile** *is the iteration space traversed by $Td(\mathcal{L})$. The part of an array that is referenced during the execution of an iteration tile is a* **data tile**. *A* **tile set** *is the union of the data tiles of all the arrays accessed during an iteration tile execution.*

A matrix multiplication loop, and the corresponding tiled matrix multiplication loop are shown in figure 2.2. The data accessed by one execution of the tiled loops (`i`,`j` and `k`) is indicated in the three darkest shades of grey, and form the data tiles of that loop. The effect of the tiling is that reuses between consecutive iterations of the outer loop (loop `i` in figure 2.2(a)) occur closer together. Array elements `B(k,j)` are reused at the outer loop level. In the tiled version, most of these

```
do i=1,N,1
  do j=1,N,1
    do k=1,N,1
      H(i,j,k)
```

Original loop nest

$\mathcal{L}$ $\left\{\right.$ $\mathrm{Ti}(\mathcal{L})$ $\left\{\right.$

```
do II = 1,N,B1
  do JJ = 1,N,B2
    do KK = 1,N,B3
```

$\mathrm{Td}(\mathcal{L})$ $\left\{\right.$

```
do i = II,min(II+B1-1,N),1
  do j = JJ,min(JJ+B2-1,N),1
    do k = KK,min(KK+B3-1,N),1
      H(i,j,k)
```

Tiled loop nest

Figure 2.1: A tiled loop nest

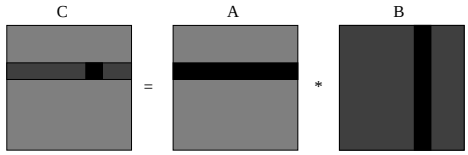
reuses occur at the `i`-loop (see figure 2.2(b)), where the number of memory accesses between use and reuse is smaller, since the number of iterations of the inner loops is limited. This is visualized in figure 2.3, where the reference distance histogram of the memory accesses before and after tiling is plotted. (The reference distance of a memory access is the number of memory accesses performed since the last access to the same data [143]). The accesses to array element `B(k,j)` generate the peak at 2^{18} in the non-tiled histogram. In the tiled version, that peak has moved to a reference distance of 2^{11}, i.e. the locality of the memory accesses to `B(k,j)` has increased.

2.2.2 Cache Memory

For the development of the cache remapping technique, a cache is represented by a tuple $(\mathsf{C_s}, \mathsf{N_s}, \mathsf{A}, \mathsf{L_s})$, following the terminology used by Ghosh [75]:

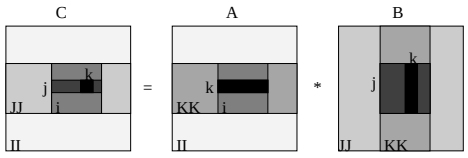
Definition 2. *The* **cache size ($\mathsf{C_s}$)** *defines the total number of bytes in the cache. The* **line size ($\mathsf{L_s}$)** *determines how many contiguous bytes are fetched from memory on a cache miss. A* **memory line** *refers to a cache-line-sized block in the memory. A* **cache set** *is the collection of cache lines in which a particular memory line can reside.* $\mathsf{N_s}$ *denotes the number of cache sets in a*

```
DO i=1,N
 DO j=1,N
  DO k=1,N
   C(i,j) += A(i,k) * B(k,j)
```



(a) Original matrix multiplication

```
DO II=1,N,B1
 DO JJ=1,N,B2
  DO KK=1,N,B3
   DO i=II,min(N,II+B1-1)
    DO j=JJ,min(N,JJ+B2-1)
     DO k=KK,min(N,KK+B3-1)
      C(i,j) += A(i,k) * B(k,j)
```



(b) Tiled matrix multiplication

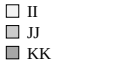


Figure 2.2: Non-tiled and tiled version of the matrix multiplication. The different shades of grey show which parts of the matrices are accessed by the different loop kernels. In the tiled code, less data is accessed to perform the same amount of computation.

cache. The **associativity (A)** *refers to the number of cache lines in a cache set. These parameters are related by the equation* $\mathsf{C_s} = \mathsf{N_s} \times \mathsf{A} \times \mathsf{L_s}$.

A mapping function computes the cache set N a memory line maps to, based on the starting address a of that memory line. In this dissertation, the standard modulo mapping function is assumed:

$$N = \left\lfloor \frac{a}{\mathsf{L_s}} \right\rfloor \bmod \mathsf{N_s} \tag{2.1}$$

After the cache set has been determined, a replacement algorithm decides into which cache line in set N the memory line is copied. In the rest of this chapter the least recently used (LRU) replacement policy is assumed.

Definition 3. *Consider the memory lines accessed during the execution of* $\mathcal{L}$*. Then* $\mathbf{Ml}(\mathcal{L}, N)$ *represents the set of memory lines which map to cache set* N*.*

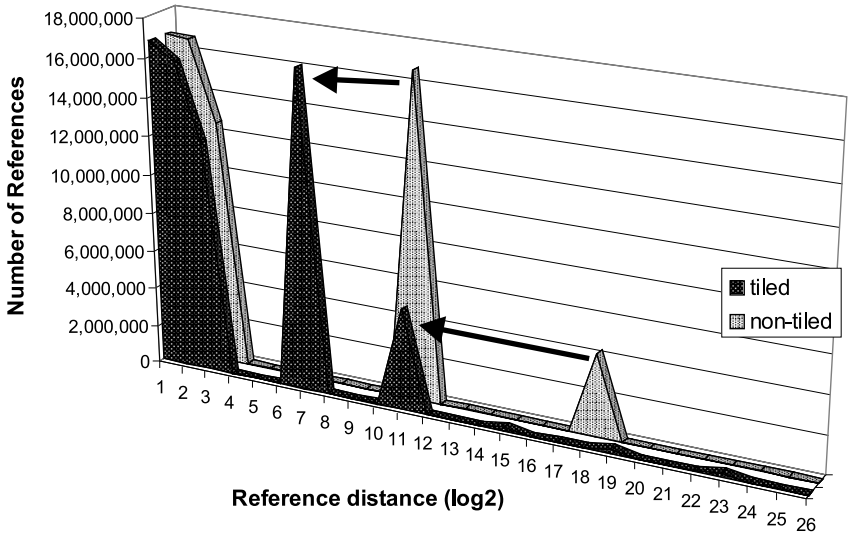


Figure 2.3: The histogram of reference distances for the matrix multiplication ($N = 256$) before and after tiling (tile size=20). From the graph, it is clear that tiling has shortened the reference distances; references with distance 2^{11} have distance 2^7 and references with distance 2^{18} have distance 2^{11} after tiling. A closer look at the graph of the tiled execution reveals a small number of references at distance $2^{15}, 2^{19}$ and 2^{23}. These result from reuses between different iterations of the tiling loops.

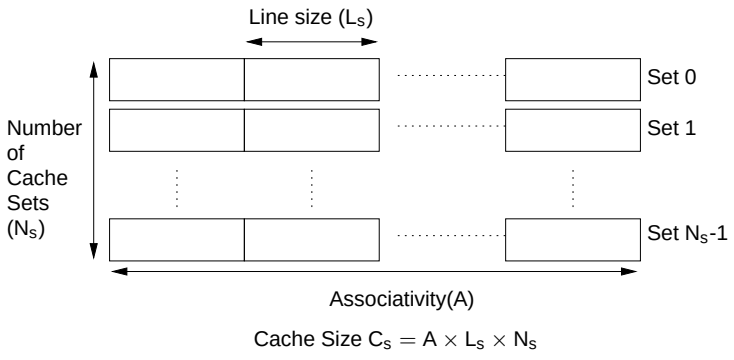


Figure 2.4: Illustration of cache organization

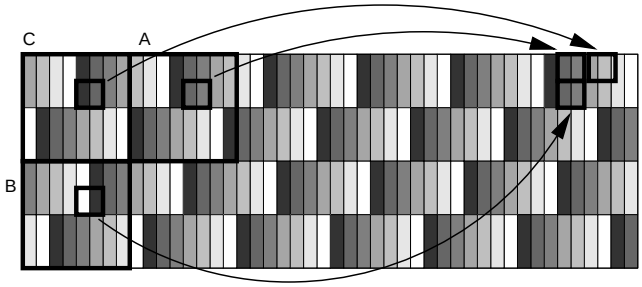


Figure 2.5: Example of a tile set in the matrix multiplication that generates conflict misses. The tiles indicated on the left occupy different memory lines that map to the same cache set (those with the same shade of grey). Cache remapping resolves this by first copying the data so that they occupy memory lines mapping to different cache sets.

2.2.3 Conflict Misses in Tiled Algorithms

Consider a tiled loop $\mathrm{Td}(\mathcal{L})$ and a cache set N. When $\#\mathrm{Ml}(\mathrm{Td}(\mathcal{L}), N) > \mathsf{A}$, more than A memory lines must be placed in the same cache set N. Only part of the memory lines can reside in the cache at the same time, and conflict misses occur.

The goal of the cache remapping method is to copy the data tiles to a new location, so that $\forall N \in \mathsf{N_s} : \#\mathrm{Ml}(\mathrm{Td}(\mathcal{L}), N) \leq \mathsf{A}$ and no conflict misses arise. An example is shown in figure 2.5. Taking into account the modulo mapping in equation (2.1), this constraint is satisfied when all tiles are copied into a contiguous $\mathsf{C_s}$-sized buffer. The goal of cache remapping is to reduce the overhead of the tile copying, by performing useful computations in parallel.

2.2.4 High-Level View of Cache Remapping

Cache remapping adds a remap thread to the program, which executes concurrently with the original processing thread executing the tiled loop nest $\mathcal{L}$ (see figure 2.6). These two threads can execute in parallel on processors with multiple functional units. (For further detail, see section 2.2.5).

Consider an iteration point i of $\mathrm{Ti}(\mathcal{L})$. The two threads work in a pipelined way (see figure 2.7):

- The *processing thread* executes tile i.

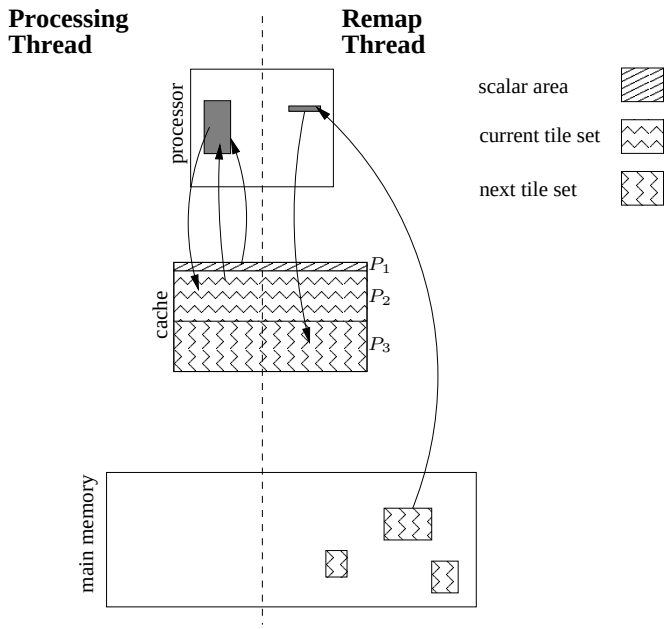


Figure 2.6: The remap thread puts the next tile set in the cache while the original thread processes the current tile set. In the next phase, the processing and the remap thread will access P_3 and P_2 respectively.

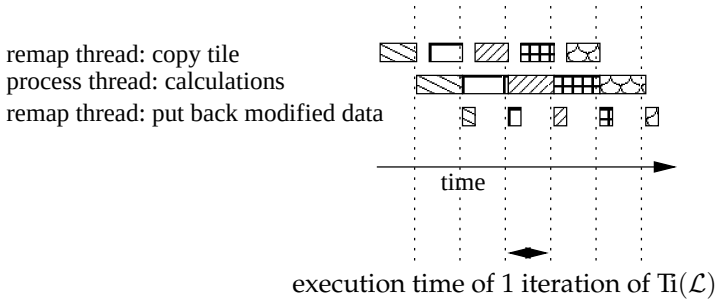


Figure 2.7: The pipelined nature of cache remapping

- The *remap thread* copies data tile set $i + 1$ into the cache. If there is changed data of tile set $i - 1$ in the cache, it is first copied back to main memory to make place for tile set $i + 1$.

At most two consecutive tile sets are in the cache at the same time. Between iterations of Ti($\mathcal{L}$), the two threads synchronize.

Fitting the Current and Next Tile Set in the Cache

The process thread accesses two kinds of variables in the memory: scalar variables which do not fit into the registers, and arrays. To ensure that all data referenced by the process thread is in the cache, it is logically divided in three parts: P_1, P_2 and P_3. P_1 is used to cache the scalars. P_2 and P_3 will each contain one tile set.

During the odd iterations of Ti($\mathcal{L}$), the process thread uses P_2, during the even iterations, it accesses P_3. The remap thread uses P_3 during the odd iterations and P_2 during the even iterations. It is clear that P_2 and P_3 must have the same size as they are used symmetrically.

Respecting Data Dependencies

Problems arise when there are data dependencies between the tile sets of two consecutive iteration tiles.

If the process thread currently processes tile set i and writes into elements of tile set $i + 1$, the remap thread prefetches these elements into the cache with the old values. When the process thread executes tile $i + 1$, it will use the old values instead of the new.

A solution is to copy the new value of the shared elements to the

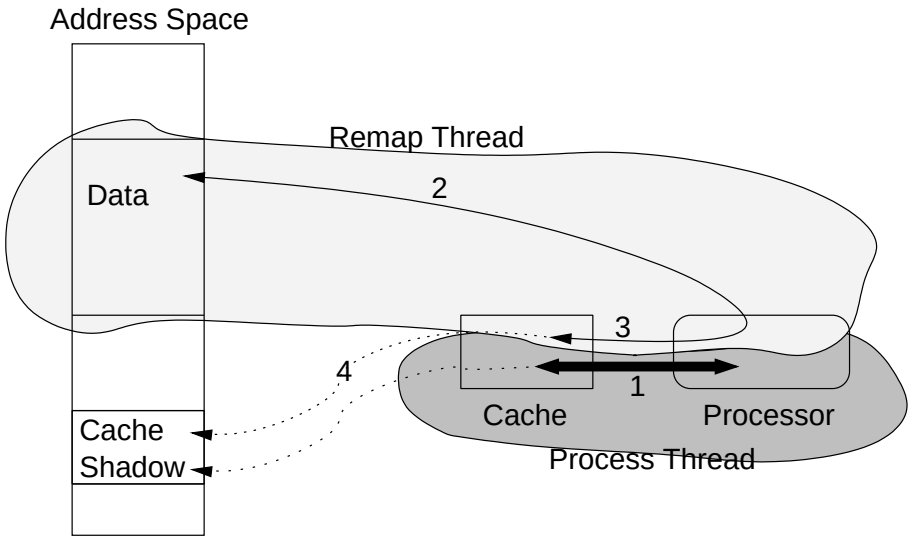


Figure 2.8: The process thread only accesses data in the cache shadow, which guarantees cache hits. The remap thread has the responsibility of copying data in the cache shadow, so that the process thread can perform computation on it.

cache partition the process thread will use. This must be done during the thread synchronization, which occurs between iterations of $\mathrm{Ti}(\mathcal{L})$.

2.2.5 Low-Level Details

Controlling Cache Behavior

The cache remapping method aims at controlling the contents of the cache by software. However, the cache hardware cannot directly be steered by software instructions. Therefore, an indirect method is needed to control the cache contents. In the cache remapping method a part of the address space with the same size as the cache is reserved. The basic idea is that only the addresses in this address space (which we name the cache shadow) are allowed to enter the cache. A conceptual picture of the cache shadow is given in figure 2.8.

Cache Shadow At the start of the program, a consecutive block of memory with size C_s is allocated and aligned on a memory line. We call this memory block the cache shadow. There's a one-to-one relation

between the addresses in the cache shadow and the storage area in the cache. The areas P_1, P_2 and P_3 are allocated in this cache shadow.

To assure that the contents of the cache shadow always resides in the cache completely, cache hints are used. They make it possible to only cache the addresses in the cache shadow by bypassing the cache on memory references outside the cache shadow.

Cache Hints In modern instruction sets, cache hints [86, 93] are attached to load and store instructions. They specify if the referred data should be cached or not. When data is loaded from/stored to P_1, P_2 or P_3, a cache hint tells the processor to cache the data. If an address outside the cache shadow is referenced, the cache hint tells the processor not to cache the data.

Thread Scheduling on a Single Threaded Superscalar Processor

The process and remap threads should run concurrently. Current microprocessors offer parallelism at the instruction level (ILP). This means that only nearby instructions in one thread can be executed simultaneously. To execute the remap thread and the process thread concurrently, these two threads need to be interwoven into a single thread at compile time. The instructions of the two threads must be interleaved so that the processor can execute instructions of the two threads during the same cycle.

There are no dependencies between the remap and the process thread during the execution of $\mathrm{Td}(\mathcal{L})$. As a result, the remap thread can use the functional units that are not used by the process thread. A good optimizing compiler can schedule the instructions of both threads so that they execute simultaneously.

Overlapping Memory Access with Computation The remap thread accesses main memory. Because the two threads are interwoven into one, it is important that the memory access doesn't stall the processor. When an instruction from the remap thread accesses main memory, there are enough independent instructions from the process thread ahead in the instruction stream to perform useful in-cache computations to overlap the latency.

```
remapA(int iter,A,p) {
  i1 = de_coalesce(iter);
  i2 = de_coalesce(iter);
  remap(p+i1*B2+i2, A[i1+II,i2+JJ]);
}
```

Figure 2.9: One of the Q functions that remap one element

Selection of Tile Size The tile size (B_1,...,B_n; $n = 3$ in the example) is chosen so that every tile set fits in P_2 and P_3. A large number of tile sizes satisfy this constraint. Let $iter_p = B_1 \times \cdots \times B_n$, the number of iterations executed by the process thread during a tile execution. Let $iter_r$ be the number of array elements that need to be remapped or put back during a tile execution. We choose to optimize the tile sizes so that the ratio $r = \frac{iter_p}{iter_r}$ is maximal. This choice assures that the processing power needed by the remap thread is as small as possible relative to the processing power needed by the process thread.

Loop Transformations and Thread Scheduling To lower the scheduling overhead, a number of loop transformations are performed to the loop nests in both threads. The remap thread originally consists of Q loop nests. Every loop nest remaps or puts back a data tile. Q_i^r is the number of elements that are remapped by loop nest i. Each of these loop nests are coalesced [137], and the body of the remaining loop is placed in an inlined remapping function (e.g. `remapA` in figure 2.9).

The innermost loop in the tiled loop nest Td($\mathcal{L}$) is unrolled $\lfloor r \rfloor$ times, then a remap call is inserted (see figure 2.11).

It is known at compile time how many times each remap function must be executed per iteration of Td($\mathcal{L}$). The outermost loop of Td($\mathcal{L}$) is split into Q parts. In each part, another remap function is called. The number of iterations in each part is chosen so that every remapping function is called at least Q_i^r times. So $Q_i^{B_1} \times B2 \times \left\lfloor \frac{B3}{r} \right\rfloor \geq Q_i^r$.

2.3 Implementation and Results

2.3.1 Processor Requirements

Three conditions must be met to enable efficient cache remapping:

```
remap(double* x, double* y)
{
   ldfd.nta  r1,y
   stfd.t1   x,r1
}
```

Figure 2.10: The remap function. The `nta` cache hint means "don't cache", the `t1` cache hint means "put into L1 cache".

```
swap(p2,p3)
iter=0
do i = II,II+Q_1^{B_1}-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      H(i,j,k,p2) /* body r */
      ...         /* times unrolled */
      H(i,j,k+r-1,p2)
                  /* remap code */
      remapA(iter++,A,p3)
iter=0
do i = II+Q_1^{B_1},II+Q_1^{B_1}+Q_2^{B_1}-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      ...
      remapB(iter++,B,p3)
...
```

Figure 2.11: The program transformations to efficiently interweave and schedule both threads into one. `p2` and `p3` are the start addresses of P_2 and P_3 respectively. It is assumed that — after inlining — the instruction scheduler will move enough independent instructions between both instructions in `remap` to allow useful computations during the main memory access.

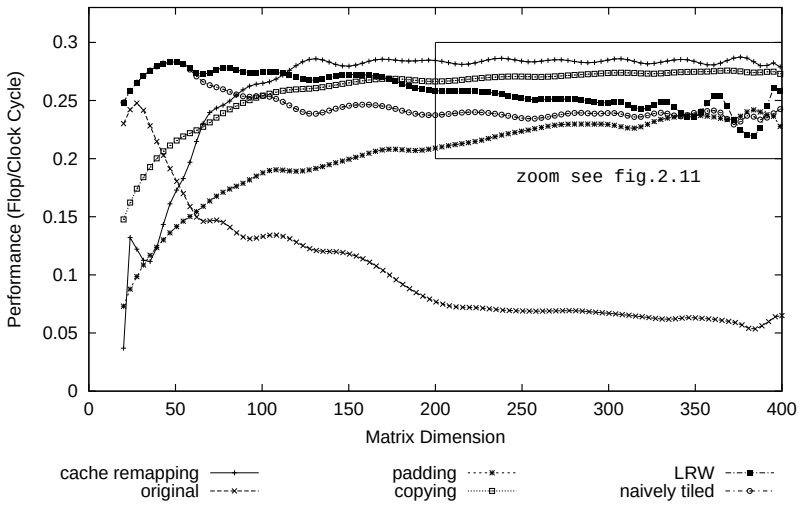


Figure 2.12: Smoothed plot of the performance of several tiled matrix multiplications for dimensions 20 to 400. In this smoothed plot it is clear that the cache remapped algorithm outperforms the others for matrix sizes bigger than 150. A zoom of the actual performance plot can be found in figure 2.13.

1. the processor provides the possibility to load data from main memory without bringing it into the cache, e.g. using cache hints,

2. multiple instructions execute concurrently, e.g. a superscalar processor,

3. the processor does not stall on a cache miss, as long as independent instructions are available in the instruction stream.

2.3.2 Simulation

The Trimaran simulator [173] was used to simulate the behavior of the processor. The cache behavior was modelled by the well known Dinero cache simulator [84].

The experiment is a tiled matrix multiplication executed on a processor with a 2-level cache. The L1 cache is 16 KB direct mapped with 32-byte lines. The L2 cache is a 256 KB 4-way set associative with 64-byte lines. We assume that the access latency of the L2 cache is 20 clock cycles and the access latency of the main memory is 65 clock cycles.

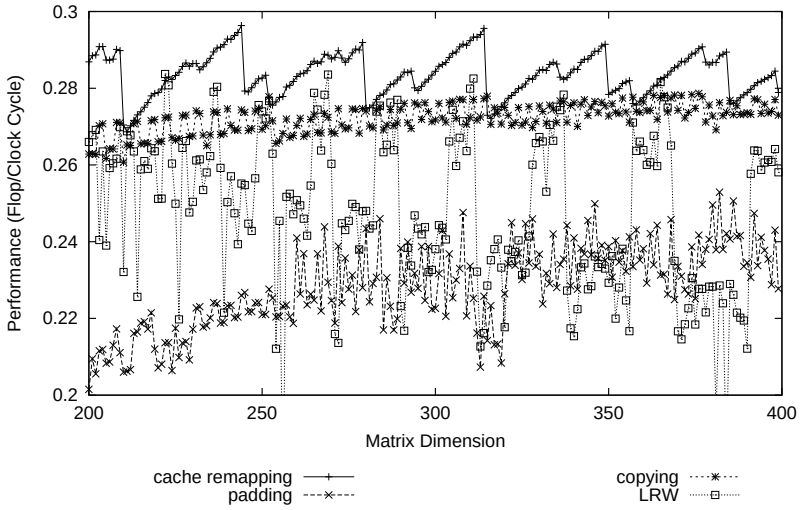


Figure 2.13: The performance of cache remapping, padding [134], copying [171] and LRW [106] on matrix dimensions 200 to 400. The cache remapped algorithm at worst has the same performance as the next best algorithm. At best, a speedup of 10% over the next best algorithm is obtained.

The cache remapping technique was compared with the original algorithm, a naively tiled algorithm not considering limited cache associativity and three optimized tiling algorithms, namely padding [134], copying [171] and LRW [106]. Each algorithm was coded, compiled and simulated for matrix dimensions between 20 and 400. For the cache remapping algorithm, the tiles on the border of the iteration space were processed using the non-pipelined copying technique [171], because the pipelined nature of cache remapping suffers from processing tiles not completely filled with data.

The performance of the algorithms, expressed in number of floating point operations per clock cycle, is plotted in figure 2.12. Because the performance of some algorithms fluctuates, the data was smoothed using Bezier curves to clearly visualize the trends. In figure 2.13 an exact plot is given for the four best algorithms for matrix dimensions 200 to 400. This plot shows that at worst, cache remapping is as good as, and at best it is 10% better than the next best algorithm.

For matrix dimensions bigger than 150, cache remapping outperforms the alternative tiled algorithms. For matrix dimensions between 200 and 400, the average speedup compared to the second best algo-

rithm (copying) is 5%. Compared with the original non-tiled algorithm, a speedup of 454% is obtained.

These results are obtained through compilation and simulation inside the Trimaran environment. On other platforms, with different compilers, larger speedups might result. For example, we would expect that a truly efficient implementation of the matrix multiplication would achieve about 1 flop per clock cycle. However, I believe that the relative performance of the different methods for removing conflict misses would remain largely the same.

2.4 Comparison with Related Work

Methods that select tile sizes to eliminate conflict misses [52, 106] sometimes result in small tiles, which reduces the performance. Padding [134] on the other hand uses large tile sizes and changes the data layout of the arrays by enlarging the dimensions with unused elements, in order to avoid cache conflicts. Unfortunately, this static adjustment cannot be optimized for every loop in a program simultaneously. Copying [106, 148, 171] eliminates conflict misses by copying the array tiles with the worst self interference to a contiguous buffer. Copying naturally involves overhead and the tradeoffs between copying and cache conflicts are discussed in [171].

In contrast to padding and tile size selection, cache remapping is independent of the array dimensions and doesn't require a change of the data layout. With respect to copying, cache remapping is able to cache tiles in a parallel thread, which runs concurrently with the processing thread. As a consequence, cache remapping has no conflict misses and incurs a minimal overhead.

The cache bypass and relocation technique was exploited by Lee [108] to use the cache as a set of vector register on i860 processors mimicking Cray's strided get/put [157]. Yamada [196] proposed prefetching and relocation by extending the hardware with a special data fetch unit which enables prefetching strided data without cache pollution. Our technique also combines cache bypass and relocation, but isn't limited to strided data patterns which allows it to prefetch and relocate data structures with non-constant strides such as data tiles.

In [159], Sen proposes a method to translate algorithms programmed for an IO-complexity model to a cache-based complexity model. Ideas

similar to copying and the cache shadow are used to explicitly control the contents of the cache from software.

At IMEC, the DTSE (Data Transfer and Storage Exploration) methodology [40] is being developed, which combines many locality and memory optimizations such as loop transformations and data placement. In contrast to cache remapping, the DTSE methodology focuses primarily on power consumption in embedded systems. Furthermore, a memory hierarchy is assumed instead of a cache hierarchy (i.e. data needs to be explicitly copied between memory levels), and it is assumed that during the optimization process, the memory parameters, such as size and number of ports are not fixed yet. This is often true in embedded systems. However, in general purpose computers, the hardware is often fixed at the time the optimization of the program begins.

2.5 Summary

This chapters begins with a short survey and categorization of the best-known software optimizations for cache behavior. Most proposed optimizations can be categorized into (1) reducing capacity misses by improving temporal or spatial locality; (2) reducing conflict misses by improving data layout or reordering computation; (3) hiding memory latency by parallel computations; (4) improving replacement policy by inserting some form of cache hints in the instructions.

The cache remapping method combines ideas from the four categories. First, cache remapping applies to tiled loop kernels. Secondly, conflict misses are resolved by relayouting data tiles at run-time. Thirdly, remaining misses are hidden by performing computation and memory fetching in two separate threads: the processing thread and the remapping thread. These threads are statically interwoven, so that they can also execute on a single-threaded processor. Finally, the placement and replacement of data is completely under control of the software, by only allowing data in the cache shadow to enter the cache; which is achieved by using cache hints.

Cache remapping was applied to a tiled matrix multiplication and compared with 4 alternative software methods to reduce the number of conflict misses in tiled loop nests. The experimental results shows that cache remapping is always at least as fast as the fastest alternative method. In the best case, cache remapping is 10% faster than the best alternative.

Chapter 3

The Reuse Distance Metric

In order to reduce the number of capacity misses, it should be understood what causes them. In this chapter, the reuse distance metric is presented, which measures the locality of memory accesses. Memory accesses with large reuse distance exhibit poor locality and result in capacity misses. The reuses are too far apart and the cache size is simply too small to capture the reuse. The reuse distance measures the cache size needed for data accesses to result in cache hits. Based on the measured reuse distances, cache hints are generated in chapter 5, and cache behavior is visualized to the programmer in chapter 6.

On contemporary systems, most programs execute hundreds of millions of memory accesses each second. A naive calculation of the reuse distance takes a very long ($O(N^2)$,N=number of memory accesses) time. In order to measure the reuse distance for a representative program execution in reasonable time, reuse distance should be calculated more intelligently. We present two methods to compute the reuse distances in a memory access trace: one is quick and exact, the other is even quicker, but only measures the reuse distance approximately.

The reuse distance distributions have been measured for the SPEC95FP programs, before and after state-of-the-art compiler optimizations, performed by a compiler that contains almost all cache optimizations proposed in the last two decades. The compiler can remove a substantial amount of misses at small reuse distances (i.e. conflict misses), but can hardly remove any misses at long reuse distance. This indicates that automatic cache optimization is hard, and further research into new methods is needed.

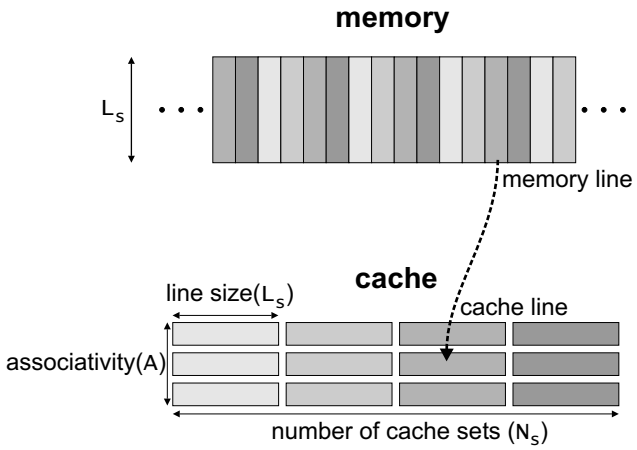


Figure 3.1: Cache model

3.1 Cache and Locality Model

To be able to talk and think about the locality of programs, how it affects cache behavior and how to improve the cache behavior, a vocabulary is needed to express common concepts [74]. In this section, a number of terms are introduced that are used throughout the rest of this dissertation to express programs, program structure, locality and the cause of cache misses.

3.1.1 Cache Terminology

C_s stands for the cache size, L_s denotes the line size of the cache, A denotes the associativity, and N_s denotes the number of cache sets. Consequently, $C_s = L_s \times A \times N_s$ [75]. The cache structure is illustrated in figure 3.1. During a memory access, the memory line on which the data lays is looked up in the corresponding cache set. If the data is present, the access is said to be a **cache hit**, otherwise a **cache miss**. In the case of a cache miss, the memory line is fetched into the corresponding cache set, where some other line is evicted according to a **replacement policy**. When not explicitly mentioned, the least recently used (LRU) policy is assumed, which evicts the line which hasn't been requested for the longest time.

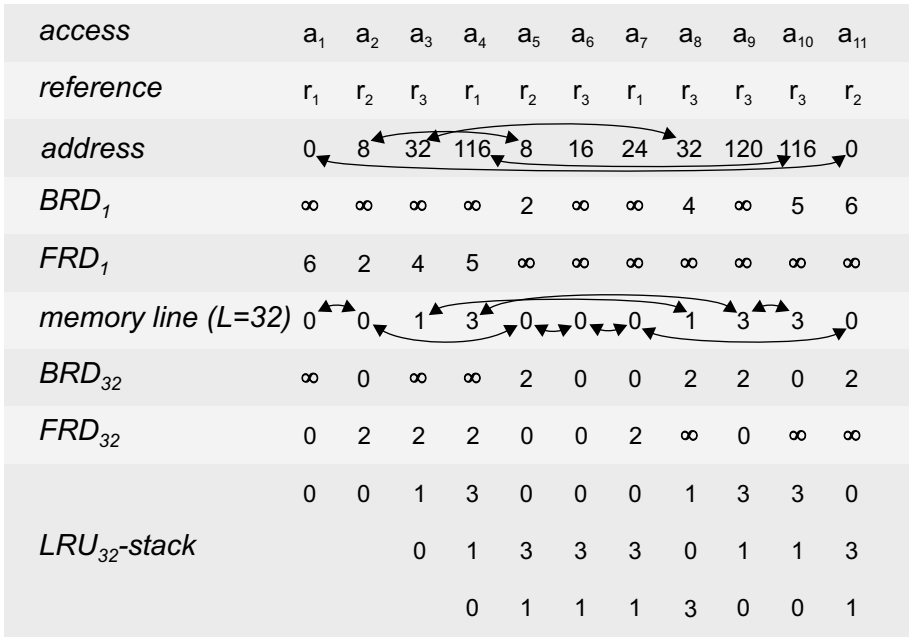


Figure 3.2: Example of a memory access stream with corresponding reuse distances. $\mathsf{RD}(\langle a_3, a_8 \rangle_1) = |\mathsf{ADS}\langle a_3, a_8 \rangle_1| = |\{8, 16, 24, 116\}| = 4$; $\mathsf{RD}(\langle a_3, a_8 \rangle_{32}) = |\mathsf{ADS}\langle a_3, a_8 \rangle_{32}| = |\{0, 3\}| = 2$.

3.1.2 Reuse Distance Terminology

First, locality and cache behavior is defined in the context of a memory access stream:

Definition 4. *A* **memory reference** *corresponds to a read or write instruction, while a particular execution of that read or write at runtime is a* **memory access***. The* **memory access stream** *is the list of all consecutive memory accesses performed during the execution of a program. A* **memory line** *is a cache-line-sized block of contiguous memory, containing the bytes that are mapped into a single cache line. [75]*

An example of a memory access stream and the corresponding references is shown in the top two rows of figure 3.2. The third row shows the address to which the access was made. The sixth row shows the corresponding memory line.

Definition 5. *A* **reuse pair** *consisting of memory accesses a_1 and a_2, denoted by $\langle a_1, a_2 \rangle_L$, is an ordered pair of memory accesses in a memory access stream,*

which touch the same memory line, without intermediate accesses to that line. The size of the memory lines is L.

If $L = 1$, the memory line consists only of the data at address a_1. As such, only temporal locality is measured. If L is larger than 1, the spatial locality that is exploited by a cache with line size L is also taken into account. The reuse pairs in figure 3.2 are indicated by arrows. In the third row, L=1. In the sixth row, L=32.

When the size of L is clear from the context, $\langle a_1, a_2 \rangle_L$ is written as $\langle a_1, a_2 \rangle$.

Definition 6. *The* **accessed data set (ADS)** *of a reuse pair $\langle a_1, a_2 \rangle$ is the set of unique memory locations accessed between a_1 and a_2, and is denoted by $\mathsf{ADS}\langle a_1, a_2 \rangle_L$.*

Definition 7. *The* **reuse distance** *of a reuse pair $\langle a_1, a_2 \rangle_L$ is the number of unique memory locations accessed between accesses a_1 and a_2. It is denoted by $\mathsf{RD}(\langle a_1, a_2 \rangle_L)$, and equals $|\mathsf{ADS}\langle a_1, a_2 \rangle_L|$.*

Definition 8. *Consider the reuse pairs $\langle a_1, a_2 \rangle_L$ and $\langle a_2, a_3 \rangle_L$. The* **forward reuse distance** *of a memory access a_2 is the reuse distance of the pair $\langle a_2, a_3 \rangle_L$. If there is no such reuse pair, its forward reuse distance is ∞. The* **backward reuse distance** *of a_2 is the reuse distance of $\langle a_1, a_2 \rangle_L$. If there is no such pair, the backward reuse distance is ∞. The forward reuse distance of a_2 is denoted by $\mathsf{FRD}_L(a_2)$, its backward reuse distance is denoted by $\mathsf{BRD}_L(a_2)$.*

The backward and forward reuse distances for the accesses in figure 3.2 are indicated for both L=1 and L=32.

When compilers optimize programs, they work with a static representation of the program, i.e. memory references instead of accesses. Therefore, it is interesting to measure the reuse distances per reference:

Definition 9. *The* **reuse distance distribution** $\mathsf{RDD}_L(r, s)$ *of a pair of references r, s is the distribution of reuse distances of all reuse pairs for which the first memory access is generated by reference r and the second memory access is generated by reference s. The* **backward reuse distance distribution of a reference** *r is the distribution of the backward reuse distance of the accesses generated by r, and is denoted by $\mathsf{BRDD}_L(r)$. The* **forward reuse distance distribution of** *r is the distribution of the forward reuse distance of the accesses generated by r and is denoted by $\mathsf{FRDD}_L(r)$.*

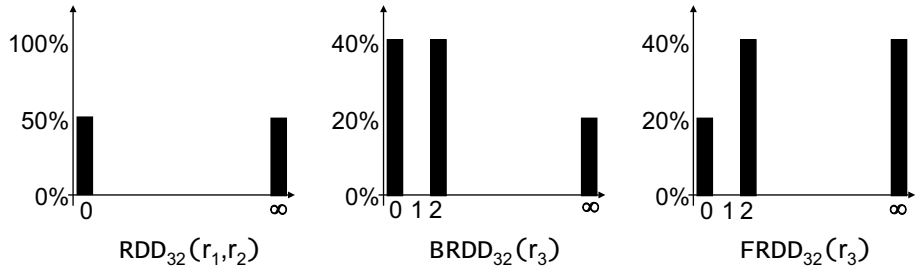


Figure 3.3: Example of backward and forward reuse distance distributions of reference r_3 in the example in figure 3.2.

Theorem 1. Reuse Distance Theorem *In a fully associative LRU cache with n lines, an access a hits the cache if and only if $\mathsf{BRD}_{\mathsf{L_s}}(a) < n$. The memory line accessed by a will stay in the cache until the next access of that memory line if and only if $\mathsf{FRD}_{\mathsf{L_s}}(a) < n$.*

Proof. In a fully associative LRU cache with n cache lines, the n most recently accessed memory lines are retained. For an access a, exactly $\mathsf{BRD}_{\mathsf{L_s}}(a)$ different memory lines were accessed since the previous access to the same location. If $\mathsf{BRD}_{\mathsf{L_s}}(a) \geq n$, the accessed memory line is not one of the n most recently accessed lines, and consequently will not be found in the cache.

If the forward reuse distance is infinite, the data will not be used in the future, so there is no next access. If the forward reuse distance is finite, consider the forward reuse distance of access a_1 and assume that the next access to the data occurs at access a_2, resulting in a reuse pair $\langle a_1, a_2 \rangle_{\mathsf{L_s}}$. By definition, $\mathsf{FRD}_{\mathsf{L_s}}(a_1)$=$\mathsf{BRD}_{\mathsf{L_s}}(a_2)$. Therefore, the data will be found in the cache at access a_2, if and only if $\mathsf{FRD}_{\mathsf{L_s}}(a_1) < n$. □

3.1.3 Relationship with Other Locality Models

Due to the importance of caches, many models have been proposed to describe cache behavior. The relation between the reuse distance and two of the most used cache behavior models (stack algorithms and the 3C's model) are discussed below.

Relation to Stack Algorithms

A replacement policy r is called a *stack algorithm* if it satisfies the inclusion property [51, 119]:

Theorem. Inclusion property for a replacement policy r

Given a memory access stream $a_1, a_2, \ldots, a_i, \ldots$*:*

memory access a_i *hits in a fully associative cache of size* $\mathsf{C_s}$

$$\Downarrow$$

memory access a_i *hits in a fully associative cache of size* $\mathsf{C_s} + 1$*.*

For stack algorithms, a single permutation of the memory lines accessed by $a_1, \ldots, a_i$ can be used to represent the contents of fully associative caches of arbitrary size after processing accesses $a_1, \ldots, a_i$. The permutation is represented as a stack, hence the name stack algorithms. The contents of the cache with size n consists simply of the top n elements in the stack.

Definition 10. *The* **LRU replacement policy** *is the policy that after each access* a *moves the memory line accessed by* a *to the top of the stack.*

Property 1. *The backward reuse distance of an access* a_i *is the depth in the LRU-stack of the requested memory line after access* a_{i-1} *has been processed.*

As an example, the LRU-stack after each memory access for caches with line size 32 is shown in figure 3.2.

Relation to 3C's Model

The most used model for explaining the cause of a cache miss is the 3C's model [85], which categorizes each miss into either a cold, a conflict or a capacity miss. Cold misses occur because the data hasn't been accessed before. Conflict misses are those misses which wouldn't occur in a fully associative LRU cache with the same size and line size, and are caused by limited associativity. Capacity misses are those that also would have occurred in a fully associative LRU cache.

Based on theorem 1, cold, conflict and capacity misses can be defined as follows:

Definition 11. *An access* a *which results in a cache miss is*

- *a* **cold miss** *if* $\mathsf{BRD}(a) = \infty$
- *a* **conflict miss** *if* $\mathsf{BRD}(a) < \mathsf{C_s}$
- *a* **capacity miss** *if* $\mathsf{C_s} \leq \mathsf{BRD}(a) < \infty$.

3.2 Fast Reuse Distance Profiling

In this section, methods for fast profiling of forward and backward reuse distances are presented. A naive algorithm is to emulate the LRU-stack using a linked list, with a list element for each memory line. On each access, the linked list can be travelled to find the requested memory line. The depth in the linked list is the backward reuse distance of that access. After the memory line has been found, the list element is moved to the top of the list. However, when N memory accesses must be profiled, this algorithm needs $O(Nd)$ accesses to list elements, where d is the average backward reuse distance. On current hardware, many programs execute hundreds of millions of memory accesses each second, so a faster method is desired. Two methods are proposed below, one which calculates reuse distance exactly, and one which is faster, but only calculates reuse distance approximately. The methods are similar to the methods presented in [130] and [101] respectively. However, they have independently been developed by the author.

3.2.1 Exact Measurement using Treaps

To enable efficient calculation of the LRU stack depth of a memory access, the stack is implemented as a *random treap* [9, 105], which is a data structure which combines a tree and a heap. A random treap is a binary tree in which each node has a key and a priority. The nodes in the tree are arranged in in-order with respect to the keys and in heap-order with respect to the priorities. By giving each node a uniform random priority, the tree is probabilistically balanced [9, 105], and the expected execution time for node insertion, deletion and lookup is $O(\log N)$.

Each node in the treap represents a memory line, and the key in each node corresponds to the stack depth of that memory line. However, the key (=stack depth) is not explicitly stored in the nodes. Instead, the stack distance is calculated as follows. Every node in the

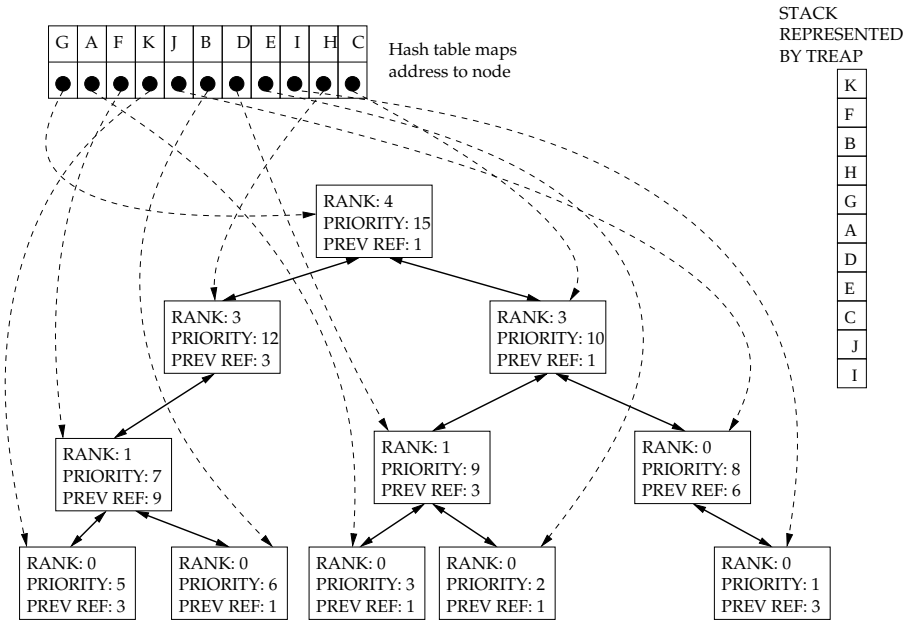


Figure 3.4: Treap representation of LRU stack after processing trace IAKJCEDFAAGHHFBFK.

treap has a rank field [102], which represents the number of nodes in its left subtree. The depth of a node N in the LRU stack is computed by walking the tree from that node upwards to the root. During the traversal, the number of nodes that are in left subtrees are summed up, provided node N is not contained in the left subtree itself. For example, in figure 3.4 the LRU stack and the equivalent treap is shown after the trace IAKJCEDFAAGHHFBFK has been processed. Suppose that address E is accessed next. The hash table is used to find the corresponding treap node (with priority 2) in constant time. To obtain the stack distance of E, the treap is walked from the node with priority 2 to the root node (with priority 15). At the first node (priority 9), a rank of 1 is found, meaning there is 1 node in its left subtree. At the second node (priority 10), the rank is ignored since the walk origins from its left subtree. The next node (priority 15) has rank 4. Summing up the rank of the nodes for which the accessed node is in the right subtree (4+1=5), and summing up the number of nodes for which the accessed node is in the right subtree (1+1=2), a LRU stack distance of 5+2=7 is obtained.

Because the tree is balanced, and has depth of $O(\log n)$, where n is

the number of memory lines in the treap, the distance can be calculated in $O(\log n)$ time. The subsequent removal of the reference node and the insertion of the node in the leftmost position in the treap also requires $O(\log n)$ time [105]. The rotation operation to maintain a balanced treap adapts the rank fields in constant time.

The stack distance found is the backward reuse distance of the current access, and the forward reuse distance of the previous access touching the same memory line. The distance is recorded in the backward reuse distance distribution of the current reference, and in the forward reuse distance distribution of the previous reference accessing the same line. The previous reference accessing the same node is stored in the PREV REF field of the corresponding treap node.

3.2.2 Approximate Measurement

Often, only an approximation of the reuse distance distribution is needed. For example, in most systems, cache sizes are powers of two, and only the behavior of these caches is of interest. For these systems, only the $\log_2$ of the reuse distance needs to be measured. This allows for a faster implementation, using the following method. The memory lines already accessed are put in a linked list, according to their LRU stack order (see figure 3.5). The node contains $\lceil \log_2(d+1) \rceil$ of its depth d in the stack. When a memory line is accessed, the hash table is used to find the corresponding node in constant time. $\lceil \log_2(d+1) \rceil$ can immediately be read from the node. After that, the stack is updated, and the node is placed on top of the stack. The nodes for which $\lceil \log_2(d+1) \rceil$ of their reuse distance changes are pointed to by the pointers in array `LOG2BORDERS`$[0..\lceil \log_2(d+1) \rceil]$. For the nodes indicated by these pointers, the $\log_2$-field is increased by 1. Finally, the pointers in the `LOG2BORDERS` array are updated. This method requires more than constant time only in the updating of the `LOG2BORDERS` array and the border nodes: $O(\log_2 d)$.

3.2.3 Program Instrumentation

Both methods described above were implemented. Furthermore, during this research, several compilers (SUIF, Pro64, ORC, FPT) have been extended to generate instrumented code which allows reuse distance measurement. The experiments in this chapter are based on an implementation in the SUIF [189] source-to-source compiler. The adapted

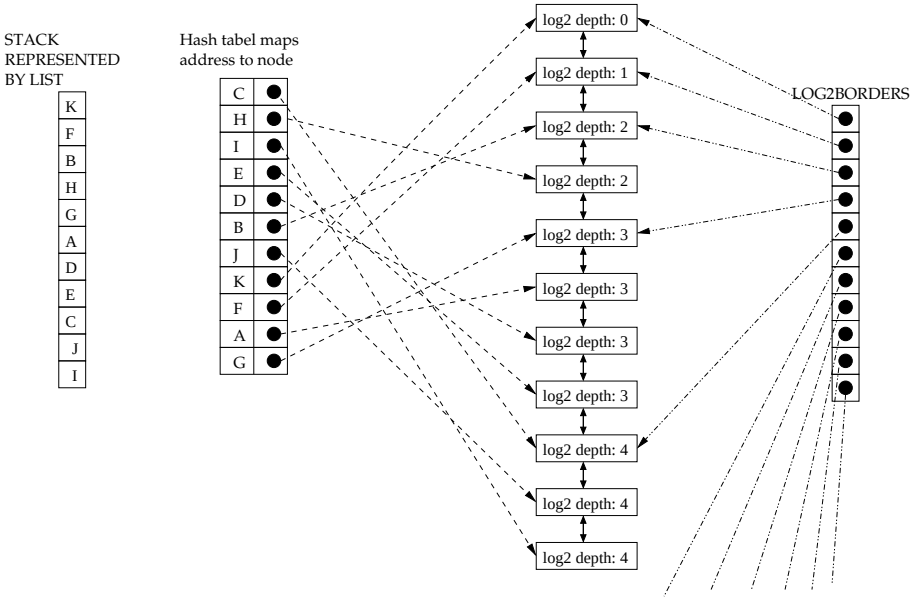


Figure 3.5: Linked list representation of LRU stack, after processing trace IAKJCEDFAAGHHFBFK, for measuring the $\log_2$ of the reuse distance in $O(\log_2 d)$ time.

compiler instruments the memory accesses in Fortran77 programs. It is assumed that all scalar variables are allocated to registers, and only for the accesses to array variables actual load and store instructions are generated after compilation. Even if some of the scalar variables cannot be allocated to registers (due to too small a register file), these variables are expected to have good locality. Consequently those accesses would result in very short reuse distance, and most likely result in cache hits. Therefore, the error made by not measuring the scalar accesses is not expected to be high.

After instrumentation, for each array reference, a function call to `measure_access` is inserted into the program with the address of the loaded data as argument. Both the treap-based method and the log2border method for calculating reuse distances have been implemented, by writing a library that implements the `measure_access` function appropriately.

The programs in the SPEC95FP [165] benchmark suite have been instrumented. The reuse distance calculation speed for these programs is plotted in figure 3.6. The figure only shows the speed of reuse distance

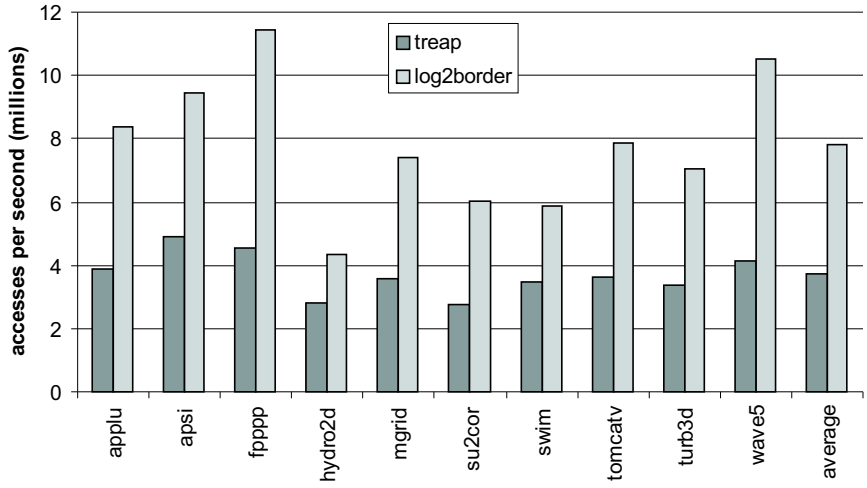


Figure 3.6: Comparison of reuse distance calculation speed using treaps and using the log2border method, as measured for the SPEC95FP programs, on a 2.66 Ghz Pentium4 processor.

calculation; the overhead of the instrumentation is not taken into account. The instrumentation overhead was measured by executing the instrumented program with an empty `measure_access`-function. The resulting execution time was subtracted from the execution times of the instrumented programs that measure reuse distance, in order to get the execution time of measuring reuse distance. It was chosen to leave out the overhead of instrumentation, to get a fairer comparison between the treap and log2border method. Furthermore, the instrumentation overhead could be reduced by instrumenting more intelligently, e.g. by only inserting an instrumentation function call at the end of each basic block.

Figure 3.6 shows that the log2border method is able to process about two times more accesses per second than the treap method. The treap-based method requires more pointers per accessed memory line, and needs more pointer manipulations to keep the data structure consistent. In the log2border method, only $\lceil \log_2(d+1) \rceil + 1$ pointers need to be updated. In the treap method, removing a node and inserting it at the left-most positions requires $O(\log d)$ rotation operations in the treap, where each rotation requires 6 pointer updates.

3.3 Effect of Compiler Optimizations on Reuse Distance and Cache Misses

In this section, we measure the reuse distance distributions of the programs in the SPEC95FP benchmark. These measurements indicate that capacity misses dominate, even for a direct mapped cache where conflict misses are expected to be the highest. Furthermore, the effect of the most important compiler cache optimizations on the reuse distance distribution and cache misses is evaluated.

3.3.1 Cache misses versus Reuse Distance

Reuse distance measures the cache misses in a fully associative cache exactly. In order to get an idea about how accurate the reuse distance is for predicting cache misses in lower-associative caches, the reuse distance of the misses in a 32 KB direct mapped cache are measured. For the SPEC95FP programs, the probability of a cache miss at a certain reuse distance is plotted in figure 3.7. The figure shows that on average, 68% of the misses are capacity misses. In the weighted average, where the weight of each benchmark is the number of executed memory accesses, 80% of the misses are capacity misses.

The ratio of conflict misses versus capacity misses depends on the cache parameters and the program. However, previous research agrees with the results in figure 3.7, and indicates that for most applications and cache configurations, capacity misses dominate [37, 83, 85, 121, 168]. Cache misses can be eliminated, either by changing the cache parameters, or by changing the program. Many optimizations have been proposed to reduce conflict misses by cache hardware optimizations [7, 41, 91, 160]. However, the only way to reduce capacity misses at the hardware level is increasing the cache size [83], which is often impossible since caches cannot be both large and fast. Therefore, capacity misses can only be eliminated by reorganizing the program. To eliminate capacity misses, the reuses at large distance must be brought closer together, so that their reuse distance becomes smaller than the cache size.

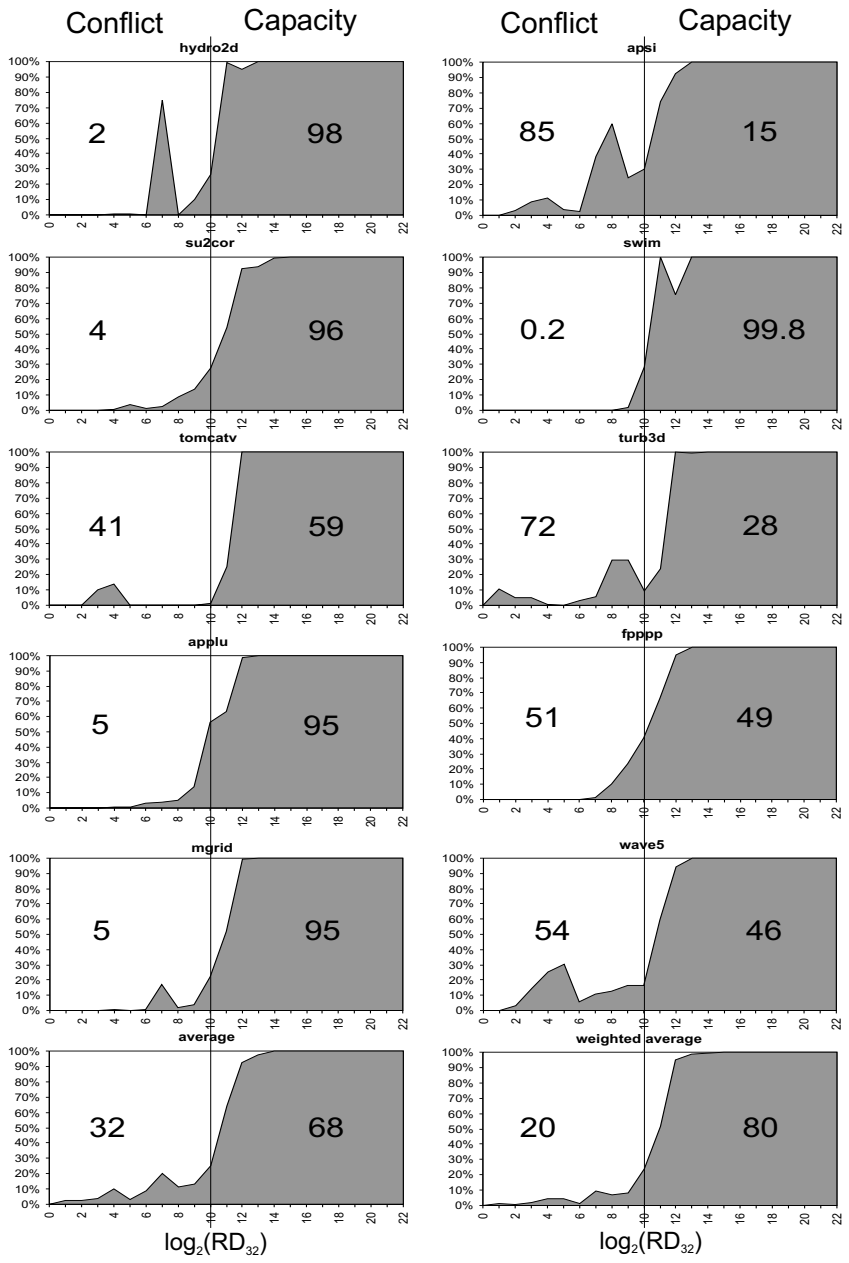


Figure 3.7: Miss rate versus reuse distance for SPEC95FP. The simulated cache is 32KB, direct mapped with 32 bytes per line. The misses with reuse distance smaller than 2^{10} are conflicts, the others are capacity misses. The percentage of conflict and capacity misses is printed on the left and on the right hand side of each plot respectively.

3.3.2 Effect of Compiler Optimization on Reuse Distance Distribution

Since capacity misses can only be reduced by program transformations, the power of automatic transformations in the SGI Pro64 compiler [161] was measured. The Pro64 compiler is an extension of SGI's MIPSpro compiler which produces IA-64 code. The compiler performs many state-of-the-art cache optimizations for both conflict and capacity misses, such as array padding, loop fusion, distribution, tiling and permutation, data shackling and others [11, 88, 103, 104, 120, 134, 146, 190, 191, 192, 193]. After the high-level cache optimizations, the Pro64 compiler can write out the transformed program as Fortran source code. The optimized source code was instrumented to get their reuse distance histograms.

The optimizations were applied to 7 of the programs in SPEC95FP. (For 3 programs, the generated source code was not legal Fortran, and could not be instrumented). In figure 3.8, the reverse cumulative frequencies[1]of the reuse distances for hydro2d, tomcatv and the overall total are shown. Hydro2d and tomcatv are the two extremes in terms of cache behavior in the set of programs. Tomcatv has a substantial number of conflict misses, which are eliminated by applying array padding. Hydro2d has only a limited number of conflict misses, which can be seen by the small difference in number of misses with reuse distance greater than 2^0 and reuse distance greater than 2^{10}.

The overall plot shows that the largest number of misses occur at reuse distances between 2^{17} and 2^{20}. Furthermore, none of these misses at long reuse distance are eliminated. In the overall total, 30% of the conflict misses are removed, while only 1.2% of the capacity misses are resolved. The plot shows that the compiler eliminates misses at relatively short reuse distances. In contrast, the compiler is unable to eliminate the misses at long reuse distance.

The misses at long reuse distance mostly occur where the use and the reuse are in different loop nests, or at different executions of the same loop nest [121]. However, most compiler loop optimizations can only handle single loop nests (e.g. loop tiling), or at most a sequence of consecutively executed loops (e.g. loop fusion and loop reversal). The fact that more conflict misses are resolved than capacity misses, agrees

[1]The reverse cumulative frequency at point x is the number of measured reuse distances larger or equal than x.

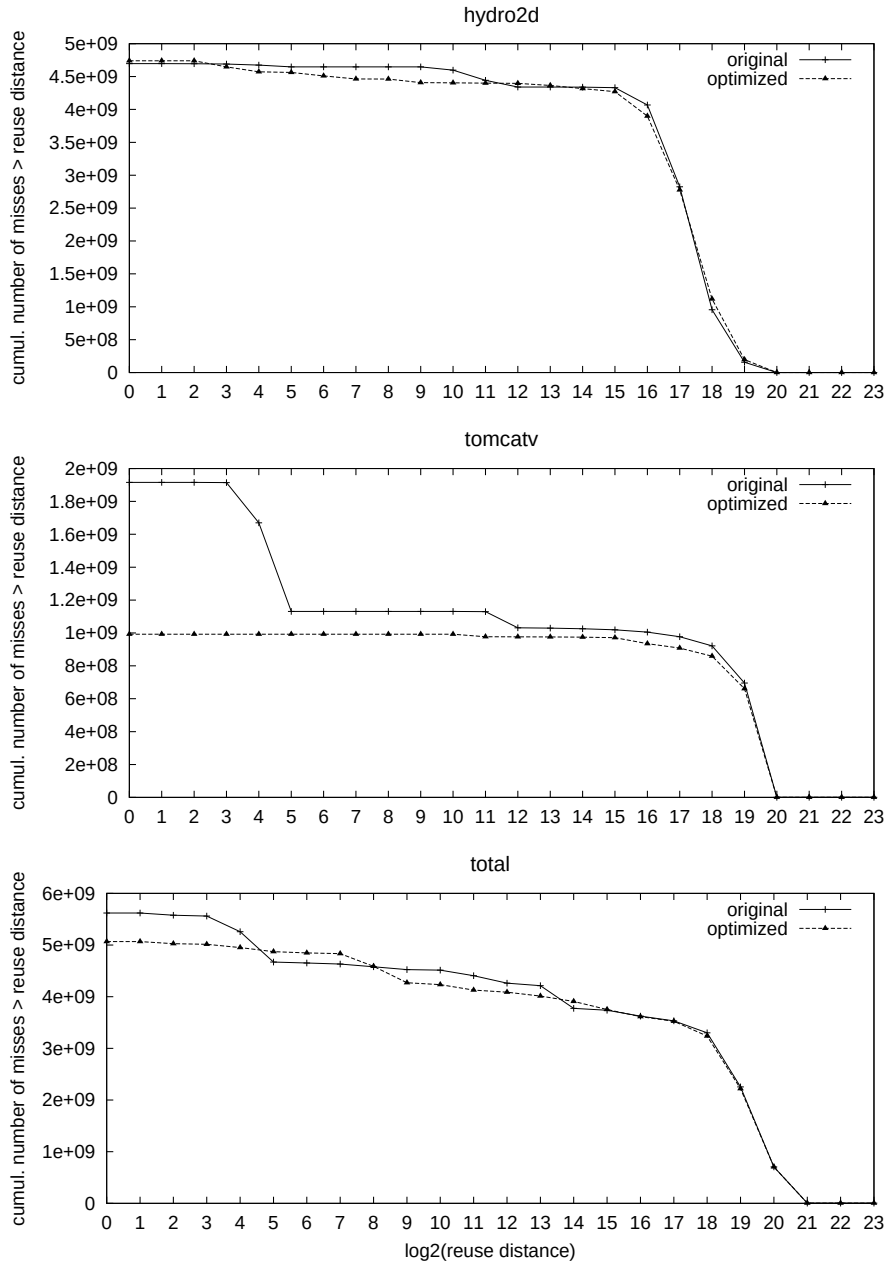


Figure 3.8: The number of cache misses before and after optimization, in function of their reuse distance. The point at reuse distance d indicates the number of misses at a distance $\geq d$.

with [121], where it is shown that most conflict misses occur in executions of single loop nests, i.e. the memory access that is prematurely evicted from the cache, and the memory access that causes the eviction occur in the same loop nest execution. As such, the compiler algorithms "see" both the original memory access and the conflicting access. On the other hand, capacity misses mostly occur between different executions of loop nests. Since most compiler optimizations only examine a single loop nest at a time, the compiler cannot observe the use and subsequent reuse. Consequently, it is not possible for the compiler to move use and reuse substantially closer together.

3.4 Related Work

It is well known that caches work because they exploit both temporal and spatial locality. However, there's no universally agreed metric to measure the amount of locality. The term "reuse distance" has first been coined by Ding [59, 60], to measure the locality of programs. Brehob [33] has shown analytically that reuse distance also predicts cache misses accurately for lower-associative caches.

Ding [62, 200] extends the cache miss prediction of programs based on reuse distance distributions, by also taking into account the effect of the program input. This is done by interpolating the reuse distance distributions of the same program, executed with different inputs.

Next to the reuse distance, the "reference distance" [143], has been proposed as a metric for locality. In contrast to the reuse distance, the reference distance counts the total number of accesses between use and reuse, not only those to unique memory locations. However, when the same data is accessed many times between two reuses, the reference distance can be very large, while the reuse distance indicates that the data is highly likely to remain in the cache. It has the advantage that it is easy and quick to measure. However, figure 3.9 shows that reuse distance calculation is only about 30% slower than reference distance calculation. In contrast to the reuse distance, there's no clear relation between reference distance and cache misses. A 30% longer computation time seems reasonable to get a clear relation between the locality metric and cache misses. McKinley and Temam [121] called this metric the "locality distance", for measuring the distance at which cache misses occur. Grimsrud [80] extends the reference distance metric, which indicates temporal locality, with spatial locality, and presents the resulting

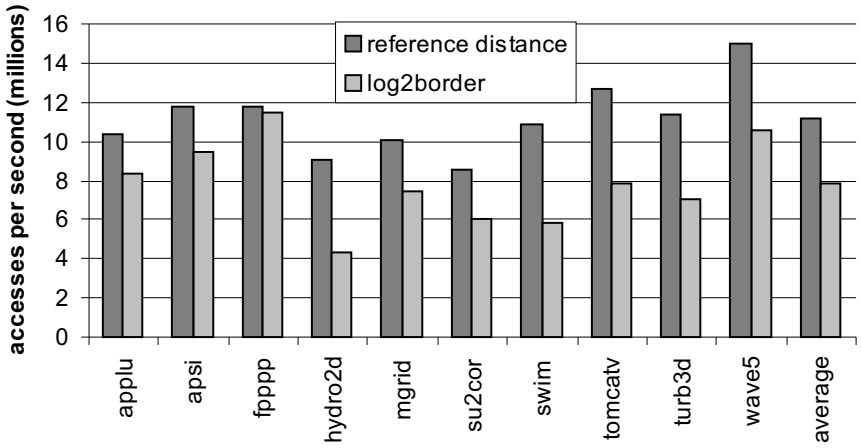


Figure 3.9: Comparison of reuse distance calculation speed and reference distance calculation speed, as measured for the SPEC95FP programs, on a 2.66 Ghz Pentium4 processor.

distributions as three-dimensional plots. From the plots, temporal locality and stride-accesses are easily recognized. However, the relation between the 3-dimensional surface plot and cache behavior is not clear.

The reuse distance is the depth of an access in the LRU stack. Representing cache contents as stacks for replacement algorithms which respect the inclusion property has first been proposed more than 30 years ago by Mattson et al. [119], in the context of page replacement in virtual memory systems [51]. The stack representations have mainly been used to allow fast simulation of a range of caches, with only a single pass over the memory access trace [85, 169, 172, 175].

Several alternative methods have been proposed to quickly calculate reuse distances, i.e. in $O(N \log d)$ time, where N is the trace length and d is the average reuse distance. Bennett et al. [17] were the first to propose a $O(N \log N)$-algorithm by using a hierarchy of vectors, resembling a tree. They need $O(N)$ space to process the trace. In contrast, more recent proposals use $O(M)$ space, where M is the number of distinct memory lines in the trace, and only need $O(N \log d)$ time. The improved efficiency is obtained by using a balanced tree representation, e.g. based on AVL trees [130], splay trees [62, 167], B-trees [199] or red-black trees [5]. Furthermore, [101] proposed a method similar

to the log2border method. Instead of representing the unique memory locations between two accesses, Almasi and Caşcaval [5, 39] indicate ranges of "holes" in the tree. A hole is an access which is not the latest access to a particular location. Recently, Ding [62] proposed approximate reuse distance calculation in $O(\log M)$ space, by compacting the tree at run-time, so that it doesn't grow larger than $4 \log_{\frac{1}{1-e}} M+4$, where the relative error on the measured reuse distance is e at most.

Next to measuring the locality in memory access streams, the reuse distance has also been employed to represent the locality in page request streams on web servers [6, 96].

3.5 Summary

The reuse distance of a reuse pair is the number of unique memory locations accessed between use and reuse. In a fully associative LRU cache, the cache misses result from reuse distances larger than the cache size. For lower associative caches, reuse distance larger than the cache size also indicates cache misses with high probability.

On current processors, programs execute hundreds of millions of memory accesses each second. In order to measure the reuse distance of all the accesses in the resulting access stream, a fast reuse distance calculation algorithm is needed. Two algorithms for measuring reuse distances are proposed. The first computes the reuse distance exactly, and uses the treap data structure to represent the LRU stack. The second method computes the reuse distance approximately (it computes $2^{\lceil \log_2(d+1) \rceil}$, where d is the reuse distance). The second method is about twice as fast as the first.

The measurement of the reuse distance distribution for the SPEC95FP benchmark indicates that the reuse distance is a good predictor for cache behavior (also for a direct mapped 32KB cache). Furthermore, it is shown that the state-of-the-art compiler optimizations can remove a large amount of the conflict misses (=misses at short reuse distances). In contrast, only a small amount of the capacity misses are eliminated. Furthermore, only the capacity misses with a reuse distance slightly larger than the cache size are eliminated. The compiler is unable to shorten any of the long reuse distances.

Chapter 4

The Reuse Distance Equations

As an alternative to costly profiling, determining the reuse distance prior to execution merely by analyzing the source code would have many benefits. This chapter introduces reuse distance equations, which result in Ehrhart polynomials which describe the reuse distance of all memory accesses as a function of the loop induction variables and program parameters. The analytical reuse distance computation eliminates the need for a costly profiling step. An additional advantage over profiling is that the resulting Ehrhart polynomials describe the reuse distance for all possible program inputs, whereas profiling measures the reuse distance distributions for one specific execution of the program.

In contrast to profiling where the billions of memory accesses are compactly represented by reuse distance distributions, the equations allow a compact representation of the reuse distance for each individual memory access.

The equations describe reuse pairs and accessed data sets by polytopes and Presburger formula. The first section discusses the necessary background on polyhedra and Presburger formulas, and how they are used to model program behavior. In the second section, the reuse distance equations are presented. The following sections present algorithms to solve the equations.

Finally, extensions to the reuse distance equations are presented, which allow to exactly describe cache behavior for caches with LRU replacement and arbitrary associativity and line size.

4.1 Polyhedral Model

The analytical calculation of reuse distances requires a framework to model program properties such as execution order and the data that is accessed between two points in time. In this work, the polyhedral model [70] is used, which describes program properties using polytopes and Presburger formula. This section introduces the necessary polyhedral theory. Furthermore, the representation of programs in the polyhedral model is discussed. The definitions in this section are often clarified by an example that follows the definition.

4.1.1 Polytopes and Polyhedra

Definition 12. *A subset P of $\mathbb{R}^n$ is called a* **convex polyhedron** *if*

$$P = \{\mathbf{x} \in \mathbb{R}^n | A\mathbf{x} \geq \mathbf{b}\},$$

where A is an $n \times c$ matrix and $\mathbf{b}$ is a $c \times 1$ column vector, i.e. P is the intersection of c half-spaces [156].

A subset P of $\mathbb{Z}^n$ is called an **integer polyhedron** *if*

$$P = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} \geq \mathbf{b}\}, \tag{4.1}$$

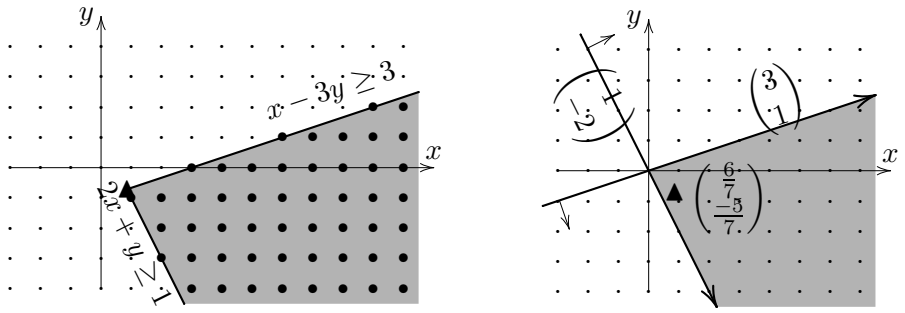
where A is an integer $n \times c$ matrix and $\mathbf{b}$ is an integer $c \times 1$ column vector [156].

Note that some authors reserve the term "integer polyhedron" to polyhedra with integer vertices, e.g. [13]. In this dissertation, "integer polyhedron" denotes the set of integer points in a polyhedron for which the matrix A and the vector $\mathbf{b}$ only contain integer values.

Example 1. *The constraints*

$$\begin{cases} 2x + y \geq 1 \\ x - 3y \geq 3 \end{cases}$$

define an integer polyhedron in $\mathbb{Z}^2$, with $A = \begin{pmatrix} 2 & 1 \\ 1 & -3 \end{pmatrix}$ and $\mathbf{b} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$. The polyhedron is geometrically represented in figure 4.1. The polyhedron can be viewed as a set of linear inequality constraints that are applied to the variables x and y. In the reuse distance equations, these variables might represent the values of the loop induction variables, or the data elements accessed at a given iteration. The constraints could be for example the loop boundaries.



(a) polyhedron

(b) Supporting cone of the vertex

Figure 4.1: Geometrical representation of the polyhedron defined by $2x + y \geq 1 \wedge x - 3y \geq 3$. The single vertex of this polyhedron is indicated by ▲. The two extremal rays coincide with the two hyperplanes defined by the inequalities.

Theorem 2. *Any polyhedron given by equation (4.1) can also be represented by its* **Minkowski representation** *[115, 156], defined by three rational matrices: a $n \times l$ matrix L, a $n \times r$ matrix R and a $n \times v$ matrix V:*

$$\left\{ \mathbf{x} \in \mathbb{Z}^n | \exists \lambda \in \mathbb{R}^l, \mu \in \mathbb{R}_+^r, \nu \in \mathbb{R}_+^n : \mathbf{x} = L\lambda + R\mu + V\nu \wedge \sum_i \nu_i = 1 \right\}, \tag{4.2}$$

where $\sum_i \nu_i = 1$ means that the sum of all elements of vector ν equals one, and $\mathbb{R}_+^r$ is the set of all r-tupels consisting of positive real numbers.

Definition 13. *The columns of matrix L of a polyhedron P in its Minkowski representation represent* **lines** *of P, the columns of R represents its* **rays**, *and the columns of V represents its* **vertices**.

Lemma 1. *Each vertex is determined by n linearly independent equations from the system $A\mathbf{x} = \mathbf{b}$ (proof: see [156]).*

Lemma 2. *The vertices of an integer polyhedron described by $A\mathbf{x} \geq \mathbf{b}$ have rational coordinates.*

Proof. This trivially follows from lemma 1 and the fact that for integer polyhedra A and $\mathbf{b}$ contain integer values. □

Example 2. *The polyhedron in figure 4.1 has one vertex: $\begin{pmatrix} \frac{6}{7} \\ \frac{-5}{7} \end{pmatrix}$. It has two rays: $\begin{pmatrix} 1 \\ -2 \end{pmatrix}$ and $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$, which are visible in figure 4.1(b).*

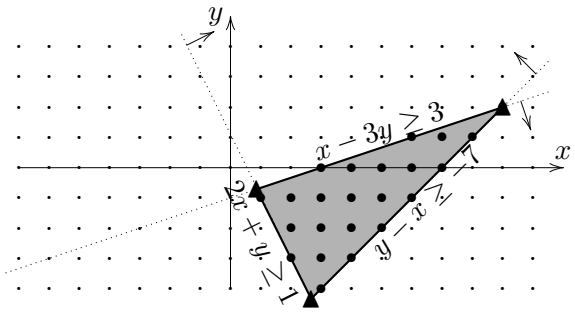


Figure 4.2: Geometrical representation of the polyhedron defined by $2x + y \geq 1 \wedge x - 3y \geq 3 \wedge y - x \geq -7$. The three vertices are indicated by ▲'s at coordinates $(\frac{6}{7}, \frac{-5}{7}), (\frac{8}{3}, \frac{-13}{3})$ and $(9, 2)$. The polytope contains no rays.

Definition 14. *A* **polytope** *is a bounded polyhedron. An* **integer polytope** *is a bounded integer polyhedron.*

Since polytopes contain a finite number of points, they have no rays nor lines. Therefore, in its Minkowski representation, it consists only of vertices. An example of an integer polyhedron is shown in figure 4.2.

Definition 15. *A set C of vectors is a* **polyhedral cone** *if*

$$C = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} \geq \mathbf{0}\}, \tag{4.3}$$

where A is a matrix [156].

Definition 16. *A cone has at most one vertex; if it has a vertex, it is located at the origin. The rays of the cone are called the* **generators** *of the cone. If the generators form a basis for $\mathbb{Z}^d$, where d is the dimension of the cone (see definition 18), then they are called* **unimodular generators**. *A cone with unimodular generators is called a* **unimodular cone**.

Definition 17. *The* **supporting cone** $\text{cone}(P, \mathbf{v})$ *of a vertex* $\mathbf{v}$ *of polyhedron P is the cone defined by*

$$\text{cone}(P, \mathbf{v}) = \{\mathbf{x} | B\mathbf{x} \geq \mathbf{0}\}, \tag{4.4}$$

where B is the submatrix of A that corresponds to the n linearly independent equations from $A\mathbf{x} = \mathbf{b}$ that define vertex $\mathbf{v}$ (see lemma 1).

Example 3. *Consider the polyhedron P in figure 4.1. The supporting cone* $\text{cone}\left(P, \begin{pmatrix} \frac{6}{7} \\ \frac{-5}{7} \end{pmatrix}\right)$ *is defined by the two linear equations that define the vertex* $\begin{pmatrix} \frac{6}{7} \\ \frac{-5}{7} \end{pmatrix}$*:*

$$\text{cone}\left(P, \begin{pmatrix} \frac{6}{7} \\ \frac{-5}{7} \end{pmatrix}\right) = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} | \begin{pmatrix} 1 & -3 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

The cone is graphically shown in figure 4.1(b). The generators of the cone are $\begin{pmatrix} 1 \\ -2 \end{pmatrix}$ *and* $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$*.*

The following definition introduces the dimension of a polyhedron (e.g. the dimension of the polytope in figure 4.3 is 1).

Definition 18. *An inequality* $\mathbf{ax} \geq \beta$ *from* $A\mathbf{x} \geq \mathbf{b}$ *is called an* **implicit equality in** $A\mathbf{x} \geq \mathbf{b}$ *if* $\mathbf{ax} = \beta$ *for all* $\mathbf{x}$ *satisfying* $A\mathbf{x} \geq \mathbf{b}$*.*

$$A^{=}\mathbf{x} \geq \mathbf{b}^{=} \text{is the system of implicit equalities in } A\mathbf{x} \geq \mathbf{b} \tag{4.5}$$

$$A^{+}\mathbf{x} \geq \mathbf{b}^{+} \text{is the system of all other inequalities in } A\mathbf{x} \geq \mathbf{b} \tag{4.6}$$

The **affine hull** *of a polyhedron* $P = \{\mathbf{x} | A\mathbf{x} \geq \mathbf{b}\}$ *is [156]:*

$$\text{aff.hull}\, P = \{\mathbf{x} | A^{=}\mathbf{x} = \mathbf{b}^{=}\}, \tag{4.7}$$

The **dimension** *of polyhedron P is the dimension of* $\text{aff.hull}\, P$*. In other words, the dimension of P is equal to n minus the rank of matrix $A^{=}$. A polyhedron of dimension k is called a k-polyhedron.*

Example 4. *Consider the polyhedron defined by* $\{(x, y) | 2x + y \geq 1 \wedge 1 \geq 2x + y \wedge -1 \leq y \leq 3\}$ *(see figure 4.3):*

$$P = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} | \begin{pmatrix} 2 & 1 \\ -2 & -1 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \geq \begin{pmatrix} 1 \\ -1 \\ -1 \\ -3 \end{pmatrix} \right\}$$

$$A^{=} = \begin{pmatrix} 2 & 1 \\ -2 & -1 \end{pmatrix},\ A^{+} = \begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix}$$

The dimension of P is $n - \text{rank}\,(A^{=}) = 2 - 1 = 1$*.*

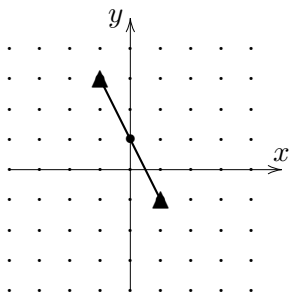


Figure 4.3: Geometrical representation of the integer polytope defined by $2x + y \geq 1 \geq 2x + y \wedge -1 \leq y \leq 3$. The dimension of the polytope is 1.

Definition 19. *F is a* **face** *of a polyhedron P if and only if F is nonempty and*

$$F = \{\mathbf{x} \in P | A'\mathbf{x} = \mathbf{b}'\} \tag{4.8}$$

for some subsystem $A'\mathbf{x} \geq \mathbf{b}'$ of $A\mathbf{x} \geq \mathbf{b}$. Each face is a polyhedron itself, and is called a k-face if it is a k-polyhedron.

Each face F of a polyhedron P is a polyhedron itself, and the faces of F are also a face of P. Therefore, the faces of a polyhedron are hierarchical, and are partially ordered.

Definition 20. *The* **face lattice** *of P is the partial ordering of the faces of P, with respect to the relation '$\subseteq$'.*

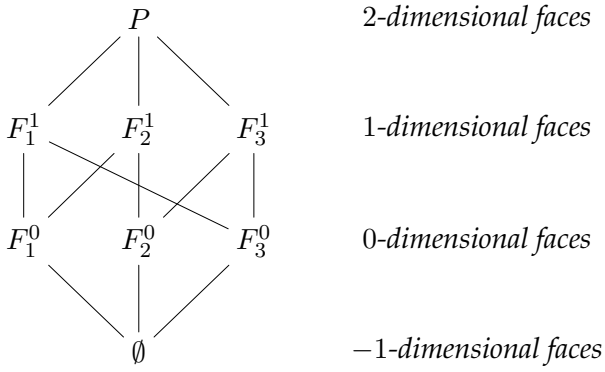
Example 5. *The polytope shown in figure 4.2 has the following 1-faces:*

$$F_1^1(P) = P \cap \{2x + y = 1\}$$
$$F_2^1(P) = P \cap \{x - 3y = 3\}$$
$$F_3^1(P) = P \cap \{x - y = 7\}$$

It has the following 0-faces:

$$F_1^0(P) = P \cap \{2x + y = 1 \wedge x - 3y = 3\}$$
$$F_2^0(P) = P \cap \{x - 3y = 3 \wedge x - y = 7\}$$
$$F_3^0(P) = P \cap \{x - y = 7 \wedge 2x + y = 1\}$$

Together with P and $\emptyset$, the faces form the following lattice under relation $\subseteq$:



Definition 21. *The* **denominator of a rational vertex** *is the lowest common multiple of the denominators of its coordinates. The* **denominator of an integer polyhedron** *is the lowest common multiple of the denominators of its vertices.*

4.1.2 Parameterized Polytopes

In a number of steps in solving the reuse distance equations (discussed later in sections 4.3– 4.5), some variables in the equations are considered to be constant parameters. This is reflected in the polyhedral model by *parameterized integer polyhedra*.

Definition 22. *A* **parameterized integer polyhedron** $P_{\mathbf{p}}$ *is a family of polyhedra, defined as*

$$P_{\mathbf{p}} = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} \geq B\mathbf{p} + \mathbf{b}\}, \mathbf{p} \in \mathbb{Z}^m \tag{4.9}$$

where A and B are constant integer matrices, $\mathbf{b}$ is a constant integer vector, and $\mathbf{p}$ is a vector of parameters [50, 115].

Example 6. *In figure 4.4, the geometrical representation is given of the following parameterized polytope:*

$$\begin{aligned} P_{\mathbf{N}} &= \{(i,j) \in \mathbb{Z}^2 : 0 \leq i \wedge 0 \leq j \wedge i + j \leq 10 \wedge i + 2j \leq N\} \\ &= \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 | \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \\ -1 & -2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ -10 \\ 0 \end{pmatrix} \right\} \end{aligned}$$

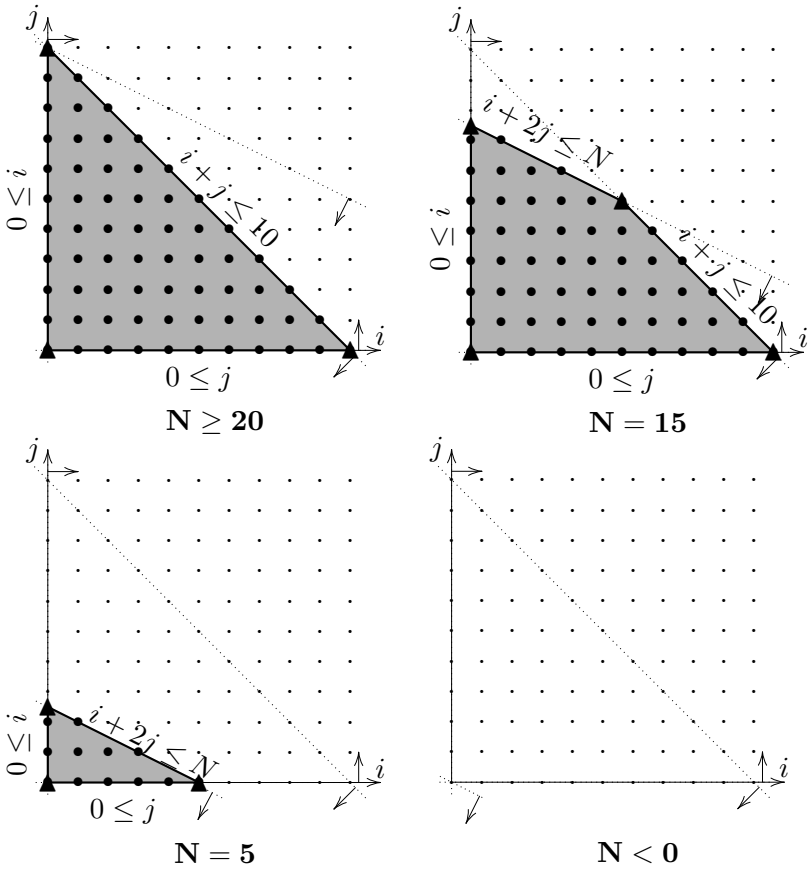


Figure 4.4: The geometrical representation of the parameterized polytope $\{(i,j) \in \mathbb{Z}^2 : 0 \leq i,j \wedge i+j \leq 10 \wedge i+2j \leq N\}$. There's a single parameter N. Notice that the number of vertices depends on the value of the parameter.

The parameterized polyhedron $P_{\mathbf{p}}$ given in equation (4.9) can be rewritten as a non-parameterized polyhedron in the combined data/-parameter space as follows:

$$P' = \left\{ \begin{pmatrix} \mathbf{x} \\ \mathbf{p} \end{pmatrix} \in \mathbb{Z}^{n+m} | A' \begin{pmatrix} \mathbf{x} \\ \mathbf{p} \end{pmatrix} \geq \mathbf{b} \right\} \tag{4.10}$$

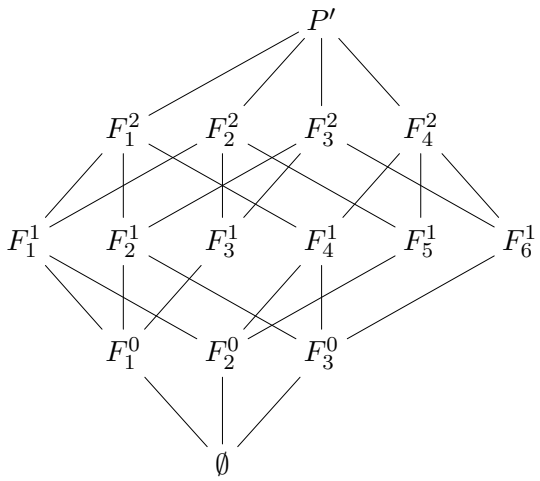
where $A' = [A|-B]$. In the reuse distance equations, polyhedra are represented mostly in this combined data/parameter space. Only when the distinction between parameters and variables is necessary, it is explicitly indicated which variables are considered parameters.

Example 7. *The parameterized polyhedron in figure 4.4, is represented in the combined data/parameter space as follows:*

$$\begin{aligned} P_{\mathbf{N}} &= \{(i,j) \in \mathbb{Z}^2 : 0 \leq i \wedge 0 \leq j \wedge i + j \leq 10 \wedge i + 2j \leq N\} \\ &= \left\{ \begin{pmatrix} i \\ j \\ N \end{pmatrix} \in \mathbb{Z}^3 | \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 0 \\ -1 & -2 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ -10 \\ 0 \end{pmatrix} \right\} \end{aligned}$$

Theorem 3. *The vertices of $P_{\mathbf{p}}$ correspond to projections of the m-faces of P', where m is the number of parameters [115].*

Example 8. *For the parameterized polytope P in figure 4.4, the parametric vertices correspond to 1-faces of P'. The faces of the combined parameter-data polyhedron are geometrically represented in figure 4.5. The faces form the following lattice:*



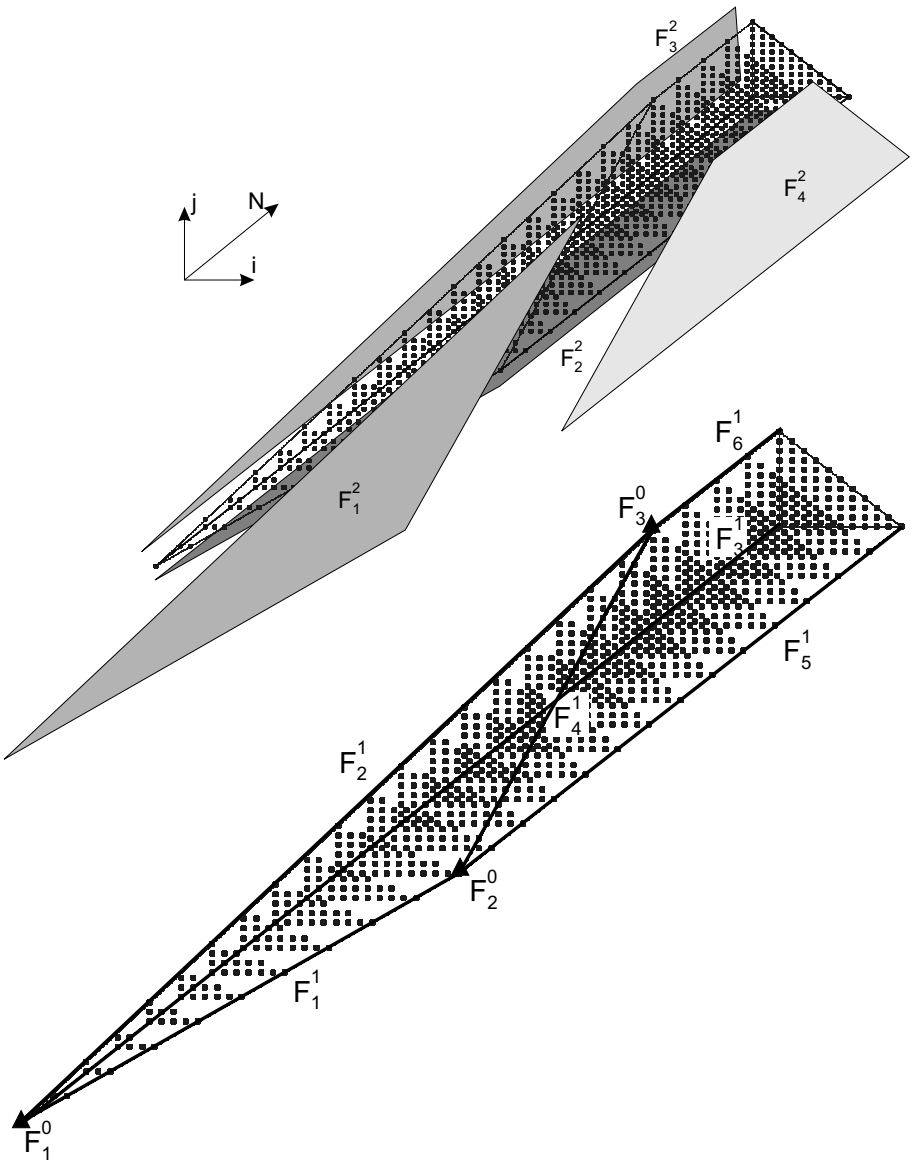


Figure 4.5: Example of combined data-parameter space, and its faces. The parametric vertices correspond to the 1-faces.

vertex	corresponding face	domain
$(N, 0)$	F_1^1	$0 \leq N \leq 10$
$(0, \frac{N}{2})$	F_2^1	$0 \leq N \leq 20$
$(0, 0)$	F_3^1	$0 \leq N$
$(20 - N, N - 10)$	F_4^1	$0 \leq N \leq 10$
$(10, 0)$	F_5^1	$10 \leq N$
$(0, 10)$	F_6^1	$20 \leq N$

Table 4.1: The parametric vertices of the polyhedron in figure 4.4.

validity domain	vertices in domain
$N < 0$	$\emptyset$
$0 \leq N \leq 10$	$(0, 0), (N, 0), (0, \frac{N}{2})$
$10 \leq N \leq 20$	$(0, 0), (0, \frac{N}{2}), (20 - N, N - 10), (10, 0)$
$20 \leq N$	$(0, 0), (10, 0), (0, 10)$

Table 4.2: The validity domains of the parametric polyhedron in figure 4.4.

Definition 23. *The* **validity domain of a parametric vertex** *is the part of the parameter space where it is defined. The* **validity domains of a parametric polyhedron** *are a partitioning of the parameter space, such that each validity domain is a maximal subset where the vertices in that domain are defined for the complete validity domain [50].*

The computation of parametric vertices is explained in detail in [115]. An algorithm for the construction of validity domains is presented in [50].

Example 9. *The parametric vertices of the polyhedron in figure 4.4 are summarized in table 4.1. The validity domains of the polyhedron are listed in table 4.2.*

4.1.3 Presburger Arithmetic

The reuse distance equations are described as Presburger formulas [139].

Definition 24. Presburger formulas *are formulas constructed from linear integer equalities and inequalities over integer variables, which are combined by logical connectives $\wedge, \vee, \neg$ and the quantifiers $\forall$ and $\exists$. An example of such a formula is $\exists y : 3x + y = 2 \wedge x < z$.*

The set corresponding to a Presburger formula consists of all integer values of the free variables that satisfy that Presburger formula. For example, the set corresponding to formula $\exists y : 3x + y = 2 \wedge x < z$ is $\{(x, z) \in \mathbb{Z}^2 | \exists y \in \mathbb{Z} : 3x + y = 2 \wedge x < z\}$.

The reuse distance computation consists of describing reuses and accessed data sets by Presburger formula. The calculation of the reuse distance of a given reuse is performed by counting the number of points in its accessed data set, which is equivalent to counting the number of solutions of the parameterized Presburger formula that describes the accessed data set. Counting the number of solutions of a parameterized Presburger formula is done by first converting it into a union of polyhedra, by the methods implemented in the Omega library [142, 195]. This conversion eliminates the $\exists$ and $\forall$ quantifiers and rewrites $\neg$-relations. The result is brought into *disjunctive normal form* (DNF) [141], i.e. the formula F has the form

$$F = \bigvee_{i=1,\ldots,n} p_{i1} \wedge p_{i2} \wedge \cdots \wedge p_{im_i}$$

where each p_{ij} is a linear integer inequality or equality. Therefore, each formula $p_{i1} \wedge p_{i2} \wedge \cdots \wedge p_{im_i}$ is a polyhedron, and F is a union of (potentially overlapping) polyhedra. This union of polytopes is converted into a disjoint union [141] of polytopes. The number of solutions of the Presburger formula is then simply the sum of the number of integer points in each of the disjoint polyhedra. Counting the number of points in polyhedra and Presburger formula is discussed in more detail in sections 4.3– 4.5.

In the rest of this chapter, Presburger formula are mostly used to model program properties. Only when the number of points satisfying a Presburger formula needs to be counted, it is assumed that the formulas are converted into a union of disjoint convex polyhedra.

4.1.4 Program Representation in the Polyhedral Model

A polyhedral model of a program describes aspects of the program by integer polyhedra and Presburger formulas. Examples of aspects which have been described previously in the polyhedral model are iteration spaces and lexicographical ordering [15, 68, 177], data dependences [53, 69, 73], memory usage [56, 114, 144], etc.

Basically, the programs for which these aspects can be modelled us-

ing only integer polyhedra and Presburger formula, satisfy the following conditions:

- The program consists of assignment, if and loop statements.
- Constant scalar variables are considered as symbolic program parameters. An example of such a parameter is `N` in figure 4.6.
- An iteration consists of a single execution of the statements in a loop. The iteration space [193] of a statement, which contains a point for every iteration in which the statement is executed, must be describable by a union of parametric integer polytopes. This is possible if the loop boundaries are linear functions of outer loop induction variables and program parameters.
- Scalars can be treated as one-dimensional arrays with size 1. The index expressions of the array variables are affine functions of the loop induction variables and program parameters.

The programs which fit the above conditions are said to fit the polyhedral model. An example program which fits the polyhedral model is shown in figure 4.6.

Definition 25.

The **set of all the references** *in a program is denoted by* $\boldsymbol{\mathcal{R}}$.

The **set of array variables** *in a program is denoted by* $\boldsymbol{\mathcal{V}}$.

The **iteration space** *of the statement in which a reference* r *occurs is denoted by* $\mathsf{IS}(\boldsymbol{r})$, *and is described by a set of integer polytopes.*

The **memory location** *which is accessed by* r *at iteration point* $i \in \mathsf{IS}(r)$ *is denoted by* $\boldsymbol{r@i}$.

The fact that iteration point i *of reference* r *is* **executed before** *iteration point* j *of reference* s *is expressed as* $\boldsymbol{i_r \ll j_s}$.

The set of program parameters is denoted by $\boldsymbol{\mathcal{P}}$.

Depending on the context, the "memory location" $r@i$ can denote the array element, the memory line, the page frame [51]. When it is not specifically indicated, it is assumed that the memory location denotes the accessed array element.

Example 10. *To clarify the notations introduced in definition 25, some examples applied to the program in figure 4.6 are given here:*

```
do i = 1, N
  A(i,i)=i
enddo
do i = 1, N
 A(2*i,1)=1
 do j = i+2, N-i
  if (i<>j) A(i,j-i) = A(3+j,i)
 enddo
enddo
```

Figure 4.6: Example program.

- *The variable set* $\mathcal{V} = \{A\}$.
- *The reference set* $\mathcal{R}$ *consists of the references whose syntactical representation is* `A(i,i)`, `A(2*i,1)`, `A(i,j-i)`, `A(3+j,i)`.
- *The iteration space* $\mathsf{IS}(\mathtt{A}(\mathtt{i},\mathtt{j}-\mathtt{i})) = \{(i,j) : (1 \leq i \leq N) \wedge (i+2 \leq j \leq N-i) \wedge \neg(i=j)\}$
- *The memory location (=array element) accessed by* `A(3+j,i)` *at iteration (i=3,j=6) is* $\mathtt{A}(3+\mathtt{j},\mathtt{i})@(i=3,j=6) = A(9,3)$.
- *The order constraint* $(i)_{\mathtt{A(2*i,1)}} \lessdot (i',j')_{\mathtt{A(i,j-i)}} \equiv i \leq i'$.
- *The parameter set* $\mathcal{P} = \{\mathtt{N}\}$.

Modelling Division and Modulo by Linear Inequalities

When modelling line-size and set-associativity, integer division and modulo operations are needed. The memory line l that contains a is $l = \lfloor \frac{a}{Ls} \rfloor$; the cache set s where l maps to is $s = l \bmod \mathsf{N_S}$. Division and modulo operations with a constant divisor can be described in the polyhedral model, by introducing an auxiliary variable [48, 141]. More formally, the term $\lfloor \frac{x}{c} \rfloor$, where c is a known constant, can be described by

$$c\alpha \leq x \leq c\alpha + c - 1. \tag{4.11}$$

In this constraint $\alpha = \lfloor \frac{x}{c} \rfloor$, as is geometrically presented in figure 4.7. In a similar way, the constraint $x \bmod c$, where c is a known constant, can be described by

$$c\alpha \leq x \leq c(\alpha+1) \wedge \beta = x - c\alpha \tag{4.12}$$

β has the same value as $x \bmod c$, as can be seen in figure 4.8.

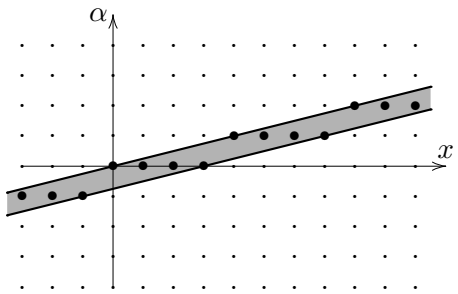


Figure 4.7: Geometric representation of division constraint $\alpha = \lfloor \frac{x}{4} \rfloor$, linearized by the constraints $4\alpha \leq x \wedge x \leq 4\alpha + 3$.

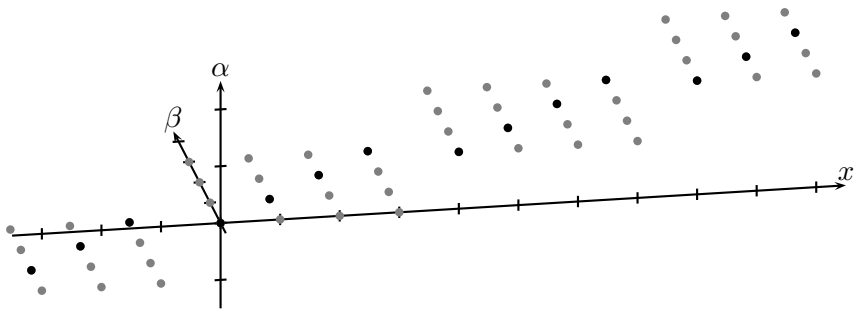


Figure 4.8: Geometric representation of modulo constraints $4\alpha \leq x \leq 4\alpha + 3 \wedge \beta = x - 4\alpha$. The dots indicate the integer points that satisfy $4\alpha \leq x \leq 4\alpha + 3$. Of these, only the black dots satisfy $\beta = x - 4\alpha$.

4.2 Reuse Distance Equations

For programs in the polyhedral model, the reuse distance of their memory accesses can be parametrically determined at compile-time. In other words, the reuse distance only depends on the program parameters and the induction variables of surrounding loops. In this section, Presburger formula are presented which allow to compute the reuse distance in a closed form.

The reuse distances of the individual accesses in the program are calculated in 3 steps:

1. The reuse pairs in the memory access stream are calculated. For every pair of *references* (r, s), $\mathsf{reuse}\,(r \to s)$ is generated, which represents all reuse pairs for which the first access is generated by an execution of reference r, and the second access is generated by s.
2. For each set of reuse pairs, a set of polytopes is constructed which describes the accessed data set (ADS) of the reuse pairs in the set.
3. The number of different memory locations in the ADS is counted, which equals the reuse distance of the reuse pair. The count is expressed by an Ehrhart polynomial.

The Presburger formulas created during the three steps are discussed below.

4.2.1 Reuse pair

First, observe the following property:

Property 2. *Every memory access is uniquely defined by the reference r which generates the access, and the iteration point I_r at which the access occurs.*

All reuse pairs $\langle x, y \rangle$ for which the first access x originates from reference r and the second access y originates from reference s, are combined into the set of reuse pairs denoted by $\mathsf{reuse}\,(r \to s)$, which contains the iteration points I_r and J_s that generate a reuse. These iteration points are described by the following simultaneous equations:

$$\forall r, s \in \mathcal{R} : \mathsf{reuse}\,(r \to s) = \{(I_r, J_s) \in \mathbb{Z}^n : \text{subject to conditions (4.14a)– (4.14d)}\} \quad (4.13)$$

$$I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s) \quad \text{(\textit{iteration space})} \tag{4.14a}$$
$$I_r \lessdot J_s \quad \text{(\textit{execution ordering})} \tag{4.14b}$$
$$r@I_r = s@J_s \quad \text{(\textit{same location})} \tag{4.14c}$$
$$\forall t \in \mathcal{R} : \neg(\exists K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s \wedge t@K_t = r@I_r) \quad \text{(\textit{no intervening access})} \tag{4.14d}$$

The above formulas specify the constraints that must be satisfied before a reuse occurs between $r@I_r$ and $s@J_s$. Equation (4.14a) expresses that I_r and J_s are part of the iteration space of r and s respectively. (4.14b) specifies that I_r must be executed before J_s; (4.14c) encodes that the same memory location must be accessed; and (4.14d) ensures that no intervening memory access touches the same memory location. Furthermore, the following formulas define the iteration points at which *forward* and *backward* reuse occurs, respectively:

$$\begin{aligned} \mathsf{reuse_F}\,(r) &= \bigcup_{\forall s \in \mathcal{R}} \{I_r : \exists J_s \in \mathsf{IS}(s) : (I_r, J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \\ \mathsf{reuse_B}\,(s) &= \bigcup_{\forall r \in \mathcal{R}} \{J_s : \exists I_r \in \mathsf{IS}(r) : (I_r, J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \end{aligned} \tag{4.15}$$

An example of the above equations for a simple program is shown in figure 4.9.

4.2.2 Accessed Data Set of a Reuse Pair

The function map_r maps an iteration space to the memory locations accessed by r, while $\mathsf{iters}_t(I_r, J_s)$ is the set of iterations of reference t executed between iteration I_r and iteration J_s:

$$\mathsf{map}_r = \{I \rightarrow r@I : I \in \mathsf{IS}(r)\} \tag{4.16}$$

$$\mathsf{iters}_t(I_r, J_s) = \{(I_r, J_s) \rightarrow K_t : K_t \in \mathsf{IS}(t) \wedge I_r \lessdot K_t \lessdot J_s\} \tag{4.17}$$

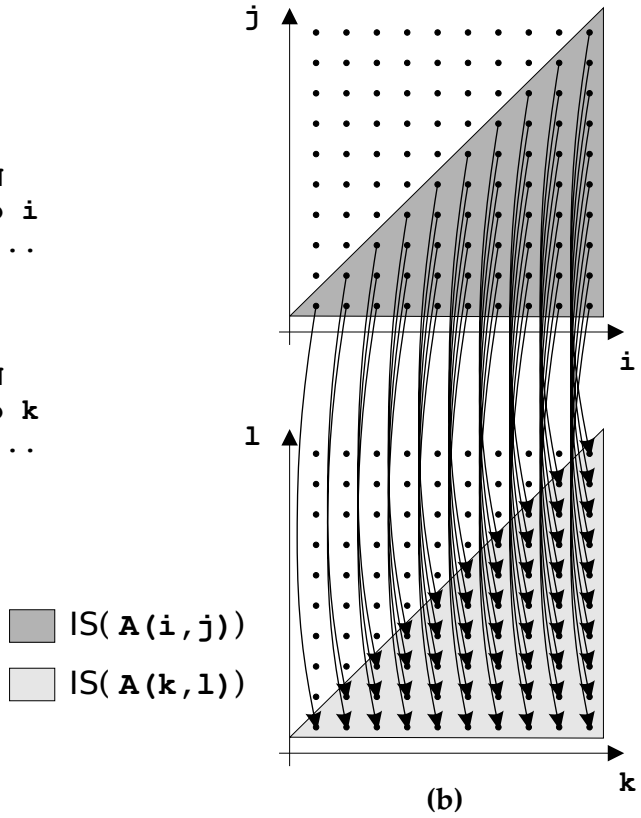
The memory locations in the accessed data set of the reuse pairs in $\mathsf{reuse}\,(r \rightarrow s)$, denoted by $\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))$, is expressed as follows:

$$\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s)) \;=\; \bigcup_{t \in \mathcal{R}} \mathsf{map}_t(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))), \tag{4.18}$$

```
for i := 1 to N
  for j := 1 to i
    A(i,j) := ...
  endfor
endfor

for k := 1 to N
  for l := 1 to k
    A(k,l) := ...
  endfor
endfor
```

(a)



(b)

(c)

- reuse ($\mathbf{A(i,j)} \rightarrow \mathbf{A(k,l)}$) = $\{(i,j,k,l) : 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge 1 \leq k \leq N \wedge 1 \leq l \leq k \wedge i = k \wedge j = l\}$
- reuse_F ($\mathbf{A(i,j)}$)= $\{(i,j) : 1 \leq i \leq N \wedge 1 \leq j \leq i\}$
- reuse_B ($\mathbf{A(i,j)}$)=$\emptyset$

Figure 4.9: The reuse pairs for a simple program. In (a), the example program is shown. In (b), the reuse pairs are shown as arrows between the iteration points of the two different references. In (c), the reuse pairs are described by integer polytopes.

Equation (4.18) expresses that the accessed data set of a reuse pair can be found by first calculating the iterations between use and reuse. Then, the ADS is simply all the data locations which are touched by the accesses in the iterations between use and reuse. The calculation of the ADS for a reuse pair of the program in figure 4.9(a) is shown in figure 4.10.

4.2.3 Reuse Distance of a Reuse Pair

In order to find the reuse distance of a reuse pair, the number of different memory locations in its ADS needs to be counted:

$$\mathsf{RD}(r,s) = \mathcal{E}(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))\,; I_r, J_s, \mathcal{P}), \tag{4.19}$$

$\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))$ is a Presburger formula that can be converted into a set of disjoint integer polyhedra. $\mathcal{E}(P;p)$ denotes the number of points in the set of integer polyhedra P, where p are the variables in the constraints that are considered parameters. The general form of the number of solutions is a set of Ehrhart polynomials (hence the notation $\mathcal{E}$), which is discussed in detail in sections 4.3, 4.4 and 4.5. $\mathsf{RD}(r,s)$ expresses the number of memory locations in a reuse pair, in function of the loop induction variables of the references r and s, and the program parameters $\mathcal{P}$.

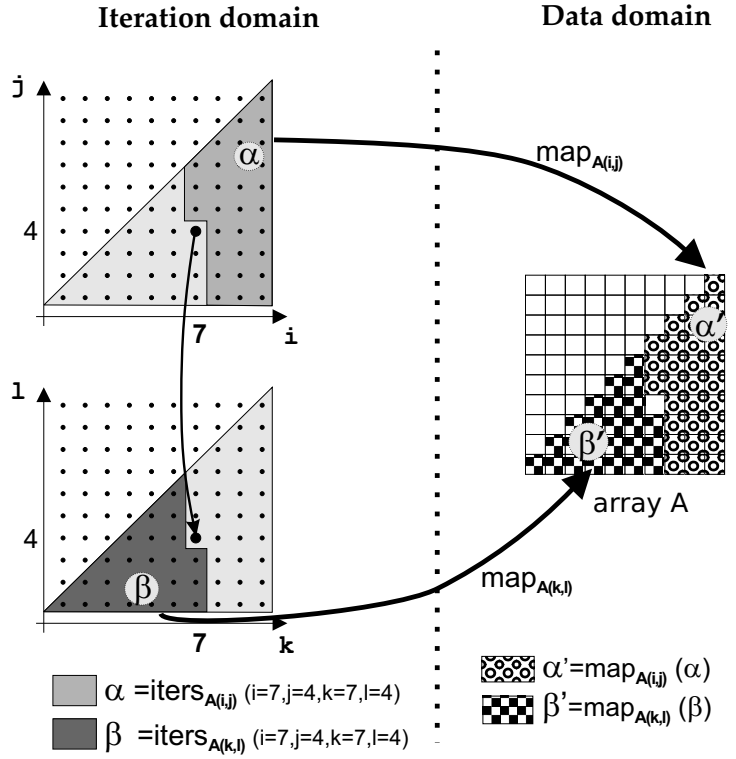
Besides calculating the reuse distance of a reuse pair, it is also possible to compute the forward and backward reuse distances of a memory reference r. These are denoted by $\mathsf{FRD}(r)$ and $\mathsf{BRD}(r)$:

$$\mathsf{FRD}(r) = \sum_{s \in \mathcal{R}} \mathcal{E}(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))\,; I_r, \mathcal{P}) \tag{4.20}$$

$$\mathsf{BRD}(s) = \sum_{r \in \mathcal{R}} \mathcal{E}(\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))\,; J_s, \mathcal{P}) \tag{4.21}$$

Furthermore, Theorem 1 on page 93 can be used to calculate at which iteration points the data will not be found in the cache and for which iteration points the data will not be retained in the cache. The iteration points where no backward reuse occurs are those where the data is fetched for the first time, resulting in a cold miss. The iteration points at which a cold miss occurs for reference r are denoted by $\mathsf{COLDM}(r)$:

$$\mathsf{COLDM}(r) = \{I : I \in \mathsf{IS}(r) \wedge I \notin \mathsf{reuse}_{\mathsf{B}}\,(r)\} \tag{4.22}$$



(a)

$$
\begin{aligned}
&\mathsf{ADS}\,(\mathsf{reuse}\,(\mathtt{A(i,j)} \rightarrow \mathtt{A(k,l)})) = \\
&\big\{(x,y) : \big(1 \le i \le N \wedge 1 \le j \le i \wedge && \mathsf{IS}(\mathtt{A(i,j)}) \\
&\qquad 1 \le k \le N \wedge 1 \le l \le k \wedge && \mathsf{IS}(\mathtt{A(k,l)}) \\
&\qquad i = k \ \wedge \ j = l\big) \wedge && \textit{same location} \\
&\qquad \big(0 \le x < i \ \vee \ x = i \wedge y < j \vee && \textit{data accessed} \\
&\qquad\ N \ge x > i \ \vee \ x = i \wedge y > j\big)\big\} && \textit{between use and reuse}
\end{aligned}
$$

(b)

Figure 4.10: Graphical representation of the calculation of $\mathsf{ADS}(\mathsf{reuse}\,(\mathtt{A(i,j)} \rightarrow \mathtt{A(k,l)}))$. A single reuse pair is shown (from reference `A(i,j)` at iteration point $(i{=}7, j{=}4)$ to the access made by reference `A(k,l)` at iteration point $(k{=}7, l{=}4)$). On the left hand side, $\alpha = \mathsf{iters}_{\mathtt{A(i,j)}}(i{=}7, j{=}4, k{=}7, l{=}4)$ and $\beta = \mathsf{iters}_{\mathtt{A(k,l)}}(i{=}7, j{=}4, k{=}7, l{=}4)$ is indicated in the iteration space. After applying the mapping functions $\mathsf{map}_{\mathtt{A(i,j)}}$ and $\mathsf{map}_{\mathtt{A(k,l)}}$, data accessed by α and β, are shown as α' and β'.

Similarly, the iteration points at which the reuse is larger than the cache size exhibits *capacity misses*, denoted by $\mathsf{CAPM}(r)$:

$$\mathsf{CAPM}(r) = \{I : \mathsf{BRD}(r) \geq \mathsf{C_s} \wedge I \in \mathsf{reuse_B}\,(r)\}, \tag{4.23}$$

The iteration points at which the accessed data will not be retained in the LRU cache can be computed as follows, and is denoted by $\mathsf{NOKEEP}(r)$:

$$\mathsf{NOKEEP}(r) = \{I : \mathsf{FRD}(r) \geq \mathsf{C_s} \wedge I \in \mathsf{reuse_F}\,(r)\}, \tag{4.24}$$

Examples of the above equations are given in figure 4.11.

4.2.4 Example: Cholesky Factorization

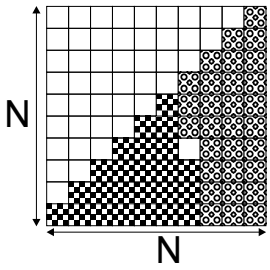
As an example of calculating the reuse distances of a real program, the Cholesky factorization code is discussed (see figure 4.12). In figure 4.13, the calculated backward reuse distances for one of the references is shown. The different reuse distance domains for the iteration space of reference `A(m,j)` is shown in the top left of figure 4.13. For each domain, the corresponding reuse distance is shown in the table in the middle. In the top right of figure 4.13, the actual backward reuse distance is shown for the different iterations. In comparison to the cumulative reuse distance distribution in the bottom left, the analytical calculation produces more detailed information: for every access, the exact reuse distance is known. Furthermore, the Ehrhart polynomials indicate the reuse distance for every possible data size, indicated by program parameter N. This is shown graphically in figure 4.14

4.2.5 Extensions to Cache Equations

The reuse distance predicts the cache behavior exactly for a fully-associative LRU cache, with a line size equal to the array element size. Other line sizes and associativities can be modelled as follows.

Modelling Larger Line Size

When multiple array elements map to the same memory line, the memory layout of multi-dimensional arrays influences the cache behavior. Also, elements from multiple arrays can map to the same memory line.



(a)

- $\mathsf{RD}(\mathtt{A(i,j)}, \mathtt{A(k,l)})$
  $= \mathcal{E}\left(\mathsf{ADS}\left(\mathtt{A(i,j)}, \mathtt{A(k,l)}\right) \; ; \; \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{N}\right)$
  $= \dfrac{N^2+N}{2} - 1$
- $\mathsf{FRD}(\mathtt{A}(\mathtt{i},\mathtt{j})) = \dfrac{N^2+N}{2} - 1$
- $\mathsf{BRD}(\mathtt{A}(\mathtt{k},\mathtt{l})) = \dfrac{N^2+N}{2} - 1$
- $\mathsf{COLDM}(\mathtt{A}(\mathtt{i},\mathtt{j})) = \{(i,j) : 1 \leq i \leq N \wedge 1 \leq j \leq i\}$
- $\mathsf{NOKEEP}(\mathtt{A}(\mathtt{i},\mathtt{j})) = \{(i,j) : 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge \dfrac{N^2+N}{2} - 1 \geq \mathsf{C_s}\}$
- $\mathsf{CAPM}(\mathtt{A}(\mathtt{k},\mathtt{l})) = \{(k,l) : 1 \leq i \leq N \wedge 1 \leq j \leq i \wedge \dfrac{N^2+N}{2} - 1 \geq \mathsf{C_s}\}$

(b)

Figure 4.11: In **(a)**, the accessed data elements of array A between use and reuse for the reuse in figure 4.10 is shown graphically. The amount of data in this set is $\frac{N^2+N}{2} - 1$, which equals the reuse distance of that reuse. In **(b)**, the forward and backward reuse distance, the iteration points where cold misses and capacity misses occur, and the iteration points for which the data will not be retained in the cache are described in function of matrix size N, using the equations (4.19)- (4.24).

```
do j = 1, N
  do l = j, N
    do k = 1, j-1
       A(l,j) = A(l,j) - A(l,k) * A(j,k)
    enddo
  enddo
  A(j,j) = sqrt(A(j,j))
  do m = j+1, N
    A(m,j) = A(m,j) / A(j,j)
  enddo
enddo
```

Figure 4.12: Cholesky factorization

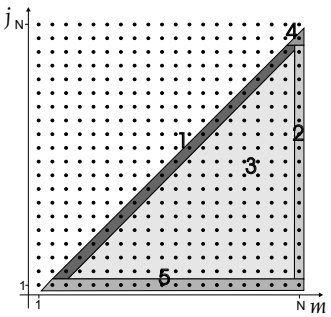
In short, the operator @ has to be defined so that it returns the memory line of a reference at a particular iteration. E.g. when the number of bytes per array element is ES, reference `A(i,j+k)` to matrix `A` with base address base_A, dimension $N \times M$ and column major order accesses memory line $\texttt{A(i,j+k)@}(i,j,k) = \left\lfloor \frac{\text{base}_A + ES \times (i + N \times (j+k))}{\mathsf{L_s}} \right\rfloor$. Notice that this only results in a Presburger formula if N is a known constant value. The division and floor operations can be transformed into a set of linear inequalities by introducing an auxiliary variable, as shown in equation (4.11). Of course, the cache size $\mathsf{C_s}$ should be expressed as the number of memory lines it can hold.

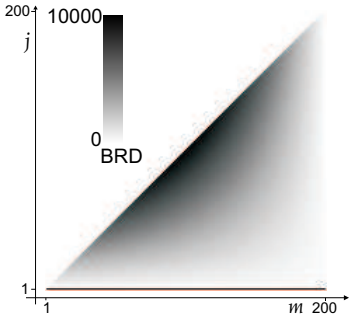
Modelling Arbitrary Associativity

The extension to set-associative caches is based on the observation that every cache set can be seen as a separate fully-associative cache, which only services memory accesses mapped to that cache set. Therefore, equation (4.18) should be adapted, so that only the memory lines that map to the same cache set between use and reuse are counted. The auxiliary function $\mathsf{set}\,(r, i)$ gives the cache set which is accessed by reference r at iteration i_r:

$$\mathsf{set}\,(r, i_r) = r@i_r \bmod \mathsf{N_s} \tag{4.25}$$

where $\mathsf{N_s}$ is the number of cache sets in the cache. The modulo-operator can be transformed into a set of linear inequalities by inserting an auxiliary variable, as shown in equation (4.12). Then, the accessed data set





nr	in domain	BRD
1	$3 \leq m = j + 1 < N$	$(N - m)j + N - 1$
2	$m = N \wedge 2 \leq j \leq N - 2$	$m - j$
3	$2 \leq j < m - 1 < N - 1$	$(N - m)j + m - 1$
4	$j + 1 = m = N \geq 3$	1
5	$1 = j < m \leq N$	∞

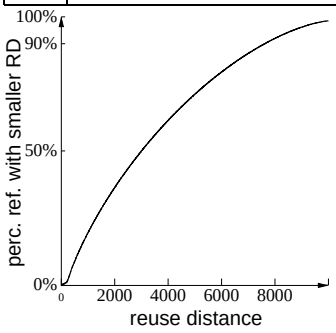


Figure 4.13: The iteration space of read reference `A(m,j)` (see figure 4.12) is shown on the top left for $N = 20$ and is divided into 5 domains. The table shows the calculated parametric backward reuse distances for the 5 domains. The calculation of these different domains is performed in detail in example 27 on page 158. In the top right, the backward reuse distance of the points in the iteration space of the same reference are shown by color, for N=200. If the pixel representing the iteration is white, the BRD is 0, when it's black, the BRD is 10000. For comparison, the cumulative reuse distance distribution is shown for this reference in the bottom left. This is the most detailed information that can be obtained by the profiling method, which clearly contains less information than the information provided by the calculated reuse polynomials.

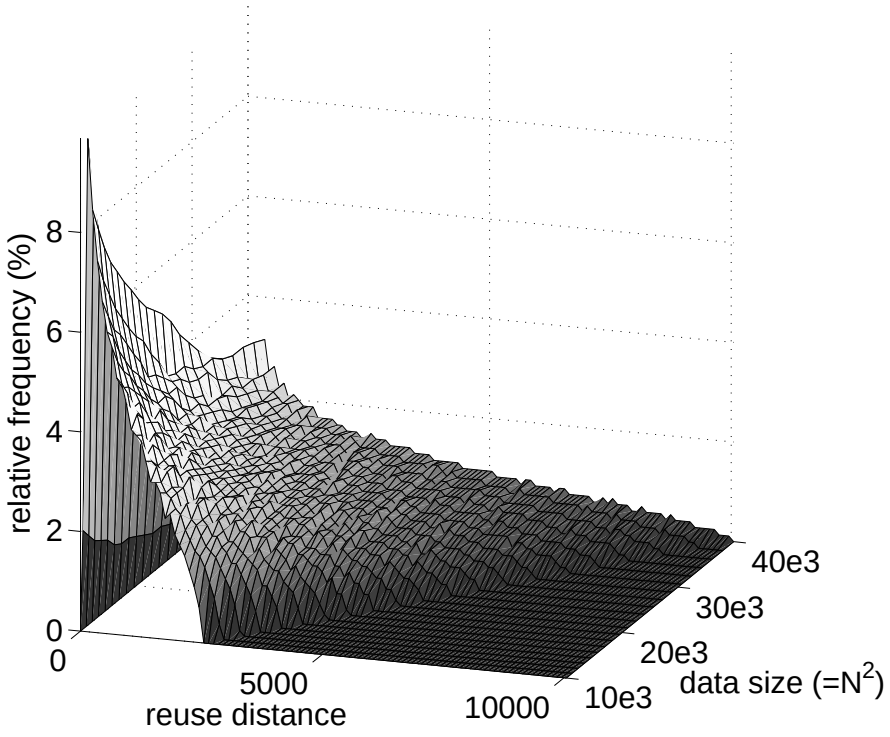


Figure 4.14: The calculated polynomials in the table in figure 4.13 specify the reuse distances for all possible data sizes, indicated by N. The reuse distance for matrix sizes for $100 \times 100 \leq N \times N \leq 200 \times 200$ is shown. When N increases, the backward reuse distances are distributed over a wider range, and the average BRD increases.

for the reuse pairs in a set-associative cache becomes:

$$\mathsf{ADS}\left(\mathsf{reuse}\left(r \rightarrow s\right)\right) = \bigcup_{t \in \mathcal{R}} \mathsf{map}_t \circ \left(\mathsf{iters}_t(\mathsf{reuse}\left(r \rightarrow s\right)) \cap \{K_t : \mathsf{set}\left(t, K_t\right) = \mathsf{set}\left(r, I_r\right)\}\right) \tag{4.26}$$

Furthermore, since each cache set is seen as a separate fully-associative cache, equations (4.23) and (4.24) should be adapted to:

$$\mathsf{MISS}(r) = \{I : \mathsf{BRD}(r) \geq \mathsf{A} \wedge I \in \mathsf{reuse}_{\mathsf{B}}\left(r\right)\}, \tag{4.27}$$

$$\mathsf{NOKEEP}(r) = \{I : \mathsf{FRD}(r) \geq \mathsf{A} \wedge I \in \mathsf{reuse}_{\mathsf{F}}\left(r\right)\}, \tag{4.28}$$

where $\mathsf{BRD}(r)$ and $\mathsf{FRD}(r)$ are the number of memory lines mapping to $\mathsf{set}\left(r, i_r\right)$ between the reuses, since equation (4.18) is replaced by equation (4.26). A is the associativity of the cache.

4.3 Enumerating Parameterized Polytopes

Section 4.2 describes how a set of parametric polytopes can be calculated which represent the addresses of the data accessed between use and reuse. To compute the reuse distance, the number of accessed memory locations between use and reuse needs to be counted, i.e. the number of integer points in the corresponding parametric polytopes needs to be computed. This section discusses counting the number of integer points in a parameterized polyhedron. In the rest of this chapter, only integer polyhedra are considered, since counting the number of integer points in polyhedra is only meaningful for integer polyhedra.

4.3.1 Ehrhart's Theory

In [65], Ehrhart discusses "des polyèdres homothétiques":

Definition 26. *A* **homothetic polyhedron** *is parametric polyhedron containing a single parameter $p \in \mathbb{Z}$, which is defined by*

$$P_{\mathbf{p}} = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} \geq \beta p\} \tag{4.29}$$

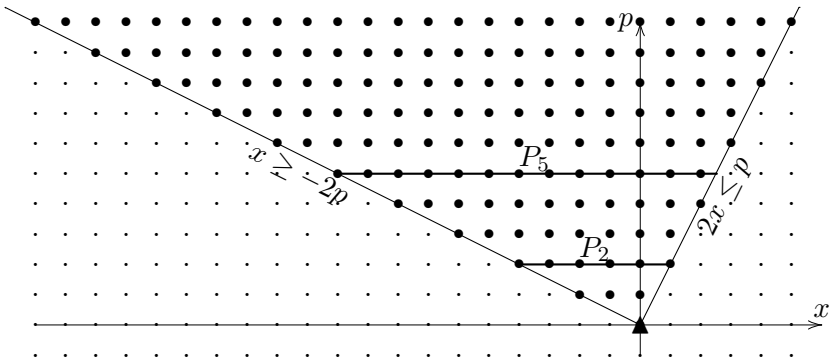


Figure 4.15: The geometrical representation of the homothetic polyhedron $P_p = \{x \in \mathbb{Z} | 2x \leq p \wedge x \geq -2p\}$, in its combined data/parameter space. All faces go through the origin. Polyhedra P_2 and P_5 are indicated by a thick black line.

Note that this is a parameterized polytope for which $\mathbf{b} = \mathbf{0}$. Therefore, all the faces of this polyhedron in the combined data/parameter space contain the origin $\mathbf{0}$, and the intersection of two faces always contains $\mathbf{0}$. Consequently, all the 1-faces contain $\mathbf{0}$. Furthermore, there's only one vertex (i.e. 0-face): $\mathbf{0}$. As a result, all 1-faces are defined for all possible values of $p \geq 0$. Taking into account Theorem 3 on page 115, it follows that the validity domain for all parametric vertices is $p \geq 0$. An example of such a homothetic polyhedron is shown in figure 4.15.

Ehrhart called the number of points in such a polyhedron "le dénombrant", or (in English) the enumerator of the homothetic polyhedron. The enumerator of a polyhedron P with parameter p is denoted by $\mathcal{E}(P; p)$. Ehrhart showed that the enumerator can be represented by a "polynôme arithmétique". In the literature, these enumerators are called Ehrhart polynomials. The coefficients of these polynomials are periodic numbers:

Definition 27. *A rational* **periodic number** $u(n)$ *is a function* $\mathbb{Z} \mapsto \mathbb{Q}$, *such that* $u(n) = u(n')$ *whenever* $n \equiv n' \pmod{p}$, $p \in \mathbb{N}$. p *is called the* **period** *of* $u(n)$.

A rational q**-periodic number** $u(\mathbf{n})$ *is a function* $\mathbb{Z}^q \mapsto \mathbb{Q}$, *such that* $u(\mathbf{n}) = u(\mathbf{n}')$ *whenever* $\mathbf{n} \equiv \mathbf{n}' \pmod{(p_1, \ldots, p_q)}$, $\mathbf{p} \in \mathbb{N}^q$.

A periodic number $u(n)$ can be represented by an array of integers

$[u_0, u_1, \ldots, u_{p-1}]_n$, where

$$u(n) = \begin{cases} u_0 & \text{if } n \bmod p = 0 \\ u_1 & \text{if } n \bmod p = 1 \\ \vdots & \\ u_{p-1} & \text{if } n \bmod p = p-1 \end{cases} \tag{4.30}$$

A q-periodic number $u(\mathbf{n})$ can be represented by a q-dimensional array $U_{(i_1,i_2,\ldots,i_q)}$ of size $(p_1, p_2, \ldots, p_q)$, such that

$$u(\mathbf{n}) = U_{(i_1,i_2,\ldots,i_q)} \quad \text{if } \forall j \in [1 \ldots q] : n_j \bmod p_j = i_j \tag{4.31}$$

Example 11. *The periodic number*

$$[1, 2]_n = \begin{cases} 1 & \text{if } n \bmod 2 = 0 \\ 2 & \text{if } n \bmod 2 = 1 \end{cases} \tag{4.32}$$

The 2-periodic number

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}_{(n_1,n_2)} = \begin{cases} 1 & \text{if } (n_1 \bmod 2, n_2 \bmod 2) = (0,0) \\ 2 & \text{if } (n_1 \bmod 2, n_2 \bmod 2) = (0,1) \\ 3 & \text{if } (n_1 \bmod 2, n_2 \bmod 2) = (1,0) \\ 4 & \text{if } (n_1 \bmod 2, n_2 \bmod 2) = (1,1) \end{cases} \tag{4.33}$$

The set of rational periodic numbers, together with their addition and multiplication, forms a ring [65]. The sum of a periodic number u_1 with period p_{u_1} and a periodic number u_2 with period p_{u_2} is a periodic number that has period $\mathrm{lcm}(p_{u_1}, p_{u_2})$ (lcm=least common multiple):

$$(u_1 + u_2)(n) = [x_0, \ldots, x_i, \ldots, x_{\mathrm{lcm}(p_{u_1},p_{u_2})-1}]_n, \text{ where } x_i = u_1(i) + u_2(i) \tag{4.34}$$

Similarly, the product of two such periodic numbers u_1 and u_2 can be computed as:

$$(u_1 \times u_2)(n) = [x_0, \ldots, x_i, \ldots, x_{\mathrm{lcm}(p_{u_1},p_{u_2})-1}]_n, \text{ where } x_i = u_1(i) \times u_2(i) \tag{4.35}$$

Definition 28. *A polynomial $f(N)$ is a* **pseudo-polynomial** *if some of its coefficients are periodic numbers instead of constants. The least common multiple of the periods of its periodic coefficients is the* **pseudo-period** *of $f(N)$.*

Furthermore, Ehrhart showed the following:

Theorem 4. Ehrhart's fundamental theorem. *The enumerator of a k-dimensional homothetic polyhedron P with parameter p is a pseudo-polynomial in p. The polynomial has degree at most k, and its pseudo-period is at most equal to the denominator of P (see definition 21). This polynomial is also called the Ehrhart polynomial.*

Example 12. *The Ehrhart polynomial describing the number of integer points for the homothetic polytope in figure 4.15 has at most degree 1 since it is a 1-dimensional parametric polytope. It has at most pseudo-period 2, since the parametric vertices are $-2p$ and $\frac{p}{2}$:*

$$\frac{5}{2}p + \left[1, \frac{1}{2}\right]_p \tag{4.36}$$

Recently, Clauss [50] extended this theorem in two ways: more than one parameter is allowed, and arbitrary parameterized polytopes are handled ($\mathbf{b}$ may be different from 0):

Theorem 5. Clauss's theorem. *The number of integer points in a parameterized k-polytope $P_{\mathbf{N}}$ is expressed at different domains of the parameter values by different multi-variable polynomials in $\mathbf{N}$ of degree k^m if the vertices of $P_{\mathbf{N}}$ are all integer points, and m is the number of parameters.*

If the vertices are rational points, the number of integer points is expressed at different domains of the parameter values by different multi-variable pseudo-polynomials in $\mathbf{N}$ of degree k^m whose pseudo-period is the denominator of $P_{\mathbf{N}}$. The parameter validity domains give the parameter ranges where the parametric vertices of $P_{\mathbf{N}}$ are defined. These polynomials or pseudo-polynomials are of the form:

$$\sum_{(i_1,i_2,\ldots,i_m)=(0,\ldots,0)}^{(k,\ldots,k)} c_{i_1,i_2,\ldots,i_m} N_1^{i_1} N_2^{i_2} \ldots N_m^{i_m} \tag{4.37}$$

where the $c_{i_1,i_2,\ldots,i_m}$ are periodic numbers whose dimensions are at most m [50].

Therefore the enumerator of a parameterized polyhedron P with parameters $\mathbf{p}$ consists of a set of pseudo-polynomials, each corresponding to a validity domain of $P_{\mathbf{N}}$. The enumerator is denoted by $\mathcal{E}(P; \mathbf{N})$. The pseudo-polynomials are also called Ehrhart polynomials.

Example 13. *The number of points in the polyhedron $P_{\mathbf{N}}$ shown in figure 4.4 is given by the following pairs of validity-domain and Ehrhart polynomials:*

$$\mathcal{E}(P;N) = \begin{cases} 0 & \textit{in validity domain} \quad N \leq 0 \\ \frac{N^2}{4} + N + \left[1, \frac{3}{4}\right]_N & \textit{in validity domain} \quad 0 \leq N \leq 10 \\ -\frac{N^2}{4} + \frac{21N}{2} + \left[-44, -\frac{177}{4}\right]_N & \textit{in validity domain} \quad 10 \leq N \leq 20 \\ 66 & \textit{in validity domain} \quad 20 \leq N \end{cases}$$

The procedure for actually computing the Ehrhart polynomials is discussed below.

4.3.2 Interpolation Method

In [49], Clauss describes how the Ehrhart polynomial can be determined, based on Theorem 5. The theorem gives the form of the solution, where only the coefficients (periodic numbers) are unknown.

Example 14. *The least common multiple of the denominators of the vertices of validity domain $10 \leq N \leq 20$ of the polyhedron in figure 4.4 is 2. The dimension of the polyhedron is 2. Therefore, Theorem 5 says that the enumerator in that validity domain has the form*

$$[a, b]_N N^2 + [c, d]_N N + [e, f]_N \tag{4.38}$$

In this form, the coefficients a, b, c, d, e and f are rational numbers which are yet unknown.

Clauss proposes to calculate the unknown coefficients by "counting some initial values of the enumerator and solving systems of symbolic rational linear equations"[49].

Example 15. *Six linearly independent equations must be found to determine the values of a, b, c, d, e and f. In the validity domain $10 \leq N \leq 20$, this can be done by fixing N at values 11, 12, 13, 14, 15 and 16, and counting the number of integer points in these non-parametric polytopes. By filling in the counts found in equation (4.38), the following linear equations are generated:*

N	*linear constraints*
11	$121b + 11d + f = 41$
12	$144a + 12c + e = 46$
13	$169b + 13d + f = 50$
14	$196a + 14c + e = 54$
15	$225b + 15d + f = 57$
16	$256a + 16c + e = 60$

After solving the above set of linear equations, the coefficients are known, and the Ehrhart polynomial is:

$$\left[-\frac{1}{4}, -\frac{1}{4}\right]_N N^2 + \left[\frac{21}{2}, \frac{21}{2}\right]_N N + \left[-44, -\frac{177}{4}\right]_N$$

The number of unknown coefficients in one single periodic number $c_{i_1,i_2,\ldots,i_m}$ is $p_1 \cdot p_2 \cdots p_m$. The number of terms in the polynomial (4.37) is $(k+1)^m$. Therefore, the number of unknown coefficients is $p_1 \cdot p_2 \cdots p_m \cdot (k+1)^m$.

4.3.3 Limitations of Interpolation Method

The interpolation method is implemented in the Polylib-library [113]. However, the interpolation method has three shortcomings that are discussed below. Afterwards, in section 4.4, an alternative method for computing the Ehrhart polynomials is described, which doesn't have the shortcomings described below.

Counting Non-Parameterized Polytopes

During the construction of the set of linear equations for computing the coefficients, the number of integer points in non-parameterized polytopes must be counted. In Polylib this is done by constructing a hyperrectangle around the polytope. Then, for each point in the rectangle it is checked whether the point satisfies all inequalities defining the polytope. If it does, the point is part of the integer polytope, and the count is increased by one. The problem is that the constructed hyperrectangle may be huge, leading to a very long run-time.

Example 16. *Consider the polytope in figure 4.4 on page 114. Assume that inequality* $i + j \leq 10$ *is replaced with* $i + j \leq 1000000$*. Then there would be a validity domain for* $1000000 \leq N \leq 2000000$*. In order to compute*

the Ehrhart polynomial for this domain, the number of points in the polytopes $P_{1000000} \ldots P_{1000006}$ *needs to be counted. The smallest enclosing hyperrectangle around* $P_{1000000}$ *is* $0 \leq i \leq 750001 \wedge 0 \leq j \leq 1000000$*. Therefore, for each of the* 750001000000 *points inside this rectangle, it must be checked if the point satisfies the constraints or not. Obviously, this leads to a long computation time.*

Until recently, no efficient method was known to count the number of integer points in a general non-parameterized polytope. Recently, a method was devised, which counts the number of points in a non-parametric polytope without resorting to enumerating all points in an enclosing hyperrectangle. The method is based on Barvinok's decomposition of cones into unimodular cones [14], and forms the basis for the improved counting method discussed in section 4.4.

Degenerate Domains

In order to compute the unknown coefficients in the periodic numbers, $(k+1)^m \cdot p_1 \cdots p_m$ *linearly independent* equality constraints need to be found. In the implementation of the method in Polylib [138], Clauss ensures linear independence by selecting the interpolation points from a hyperrectangle in the parameter space. The fact that the parameter points from a hyperrectangle of size $p_1(k+1) \times p_2(k+1) \times \cdots \times p_m(k+1)$ results in linearly independent equations can be seen as follows. Consider a parametric polytope with period p and a single parameter n. The general form of the Ehrhart polynomial EP is

$$EP = [c_{0,d}, c_{1,d}, \ldots, c_{p-1,d}]_n n^d + [c_{0,d-1}, \ldots, c_{p-1,d-1}]_n n^{d-1} + \cdots + [c_{0,0}, c_{1,0}, \ldots, c_{p-1,0}]_n \quad (4.39)$$

and it can also be viewed as a combination of p regular polynomials over $\mathbb{Q}$:

$$EP = \begin{cases} c_{0,d}n^d + c_{0,d-1}n^{d-1} + \cdots + c_{0,0} & \text{if } n \equiv 0 \mod p \\ c_{1,d}n^d + c_{1,d-1}n^{d-1} + \cdots + c_{1,0} & \text{if } n \equiv 1 \mod p \\ \vdots & \vdots \\ c_{p-1,d}n^d + c_{p-1,d-1}n^{d-1} + \cdots + c_{p-1,0} & \text{if } n \equiv p-1 \mod p \end{cases} \quad (4.40)$$

To calculate the values of $c_{i,j}$, for each polynomial in equation (4.40) (i.e. each value of n between 0 and $p-1$), the following set of linear

nr	validity domain	vertices $\binom{i}{j}$
1	$-2 \leq 4N + M \leq 2$	$\binom{5N}{-5N}$, $\binom{N-M-2}{3N+2M+4}$, $\binom{N-M-2}{-N+M+2}$
2	$2 \leq 4N + M$	$\binom{N-M+2}{3N+2M-4}$, $\binom{N-M+2}{M-N-2}$, $\binom{N-M-2}{3N+2M-4}$, $\binom{N-M-2}{M-N+2}$

Table 4.3: The validity domains of the parametric polytope in equation (4.42).

equations is constructed:

$$\begin{pmatrix} n^d & n^{d-1} & \cdots & 1 \\ (n+p)^d & (n+p)^{d-1} & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ (n+dp)^d & (n+dp)^{d-1} & \cdots & 1 \end{pmatrix} \begin{pmatrix} c_{n \bmod p,d} \\ c_{n \bmod p,d-1} \\ \vdots \\ c_{n \bmod p,0} \end{pmatrix} = \begin{pmatrix} \#P(n) \\ \#P(n+p) \\ \vdots \\ \#P(n+dp) \end{pmatrix}, \tag{4.41}$$

where $\#P(n)$ is the number of points in the non-parameterized polytope with the parameter having value n. The matrix in equation (4.41) is a Vandermonde matrix. $n, n+p, \ldots, n+dp$ are all different values, therefore, the Vandermonde determinant is different from 0, and the equations are linearly independent.

However, for some forms of the validity domains it is impossible to construct a hyperrectangle of size $(pd)^m$ which fits in the validity domain, and the method fails. Validity domains for which the Ehrhart polynomial cannot be computed through interpolation are known as "degenerate domains".

Example 17. *Consider the two-dimensional parametric polytope over the variables i and j with parameters N and M defined by the constraints*

$$\begin{cases} N - M - 2 \leq i \\ i \leq N - M + 2 \\ 0 \leq i + j \\ i + j \leq N \end{cases} \tag{4.42}$$

This parametric polytope has two validity domains, as shown in table 4.3 and

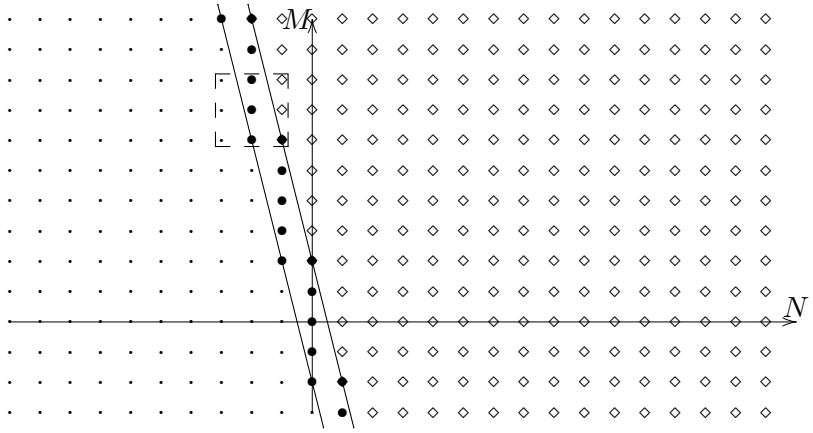


Figure 4.16: Geometrical representation of the validity domains of equation (4.42). The points in the first validity domain are indicated by •; the points in the second validity domain are indicated by ⋄. The rectangle indicated by dashed lines can never fit in the first validity domain.

in figure 4.16. The Ehrhart polynomial for validity domain 1 has the form

$$a + bN + cM + dN^2 + eNM + fM^2, \tag{4.43}$$

since the polytope is two-dimensional, and the least common denominator of the vertices is 1. The interpolation method tries to construct a rectangle of size 3×3 *which fits in the validity domain. No such rectangle exists, and the method fails. However, the Ehrhart-polynomial can be computed by considering the following parameter points:*

(M, N)	*equation*
$(0, 0)$	$a = 6$
$(1, 0)$	$a + c + f = 10$
$(2, 0)$	$a + 2c + 4f = 15$
$(2, -1)$	$a - b + 2c + d - 2e + 4f = 1$
$(3, -1)$	$a - b + 3c + d - 3e + 9f = 3$
$(6, -2)$	$a - 2b + 6c + 4d - 12e + 36f = 1$

which leads to the solution

$$6 + 14N + \frac{7}{2}M + 8N^2 + 4NM + \frac{1}{2}M^2, \tag{4.44}$$

However, it is not clear how to devise a general algorithm which can always find parameter points which lead to a set of linearly independent equations.

```
do i = 0, 199
  do j = 0, 199
    s = 0
    do k = 0, 199
      s = s + A(i,k) * B(k,j)
    enddo
    C(i,k) = s
  enddo
enddo
```

Figure 4.17: Matrix multiplication

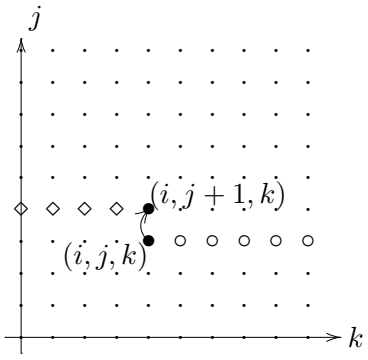


Figure 4.18: Intermediate accesses between reuses of `A(i,k)` at iterations points (i, j, k) and $(i, j + 1, k)$.

Large Periods

For some parameterized polyhedra, the period of the Ehrhart polynomial can be large, while the Ehrhart polynomial could be represented in a simpler form. This is best illustrated by an example.

Example 18. *Consider the matrix multiplication code in figure 4.17.*

Iterations (i, j, k) *and* $(i, j+1, k)$ *access the same array element* `A(i,k)`*. In order to make the limitations of the interpolation method more obvious, the number of TLB pages are counted in this example, since it leads to larger periods than when counting cache lines. The number of distinct TLB pages accessed between these two accesses can be counted as follows. For simplicity it is assumed that* `A(i,k)` *and* `B(k,j)` *access different TLB pages and only* `A(i,k)` *needs to be considered. Assume that* `A` *is a* 200×200 *matrix, which*

is layed out in column major order, and starts at address zero. Furthermore, an element size of 4 bytes is assumed. As such, `A(i,k)` *is located at address* $4 \times (200k + i)$.

Figure 4.18 shows the iterations that are executed between (i, j, k) *and* $(i, j + 1, k)$*: iterations* $(i, j, k + 1 \ldots 199)$ *(*$\circ$ *in the figure) and iterations* $(i, j + 1, 0 \ldots k - 1)$ *(*$\diamond$ *in the figure). The set of TLB pages accessed by the* $\circ$*-iterations can be described as follows*

$$S_1 = \left\{ p | \exists k' : p = \left\lfloor \frac{800k' + 4i}{L} \right\rfloor 0 \leq i, j, k \leq 199 \wedge k + 1 \leq k' \leq 199 \right\}, \tag{4.45}$$

where i*,* j *and* k *are parameters. This can be written as a set of linear constraints:*

$$S_1 = \{p | \exists k' : 1024p \leq 200k' + i \leq 1024p + 1023 \\ \wedge 0 \leq i, j, k \leq 199 \wedge k + 1 \leq k' \leq 199)\},$$

where a page size $L = 4096$ *is assumed, and is simplified to (e.g. using the Omega-library)*

$$S_1 = \{p | 0, 1024p - 39800 \leq i \leq 199 \wedge 0 \leq k \leq 198 \\ \wedge 0 \leq j \leq 199 \wedge i + 200k \leq 823 + 1024p\}.$$

S_1 *is a one-dimensional polytope with vertices*

$$\frac{i}{1024} + 25\frac{k}{128} - \frac{823}{1024} \qquad \textit{and} \qquad \frac{i}{1024} + \frac{4975}{128}. \tag{4.46}$$

Therefore, the corresponding Ehrhart polynomial has degree 1 and period 1024 in i *and period 128 in* k*. So, the number of terms in the polynomial is 4, and each has a periodic coefficient with period* 1024×128*. In total, the Ehrhart polynomial is represented by* $4 \times 1024 \times 128 = 524288$ *rational numbers. However, using the method based on Barvinok's decomposition (described later in section 4.4), the following far more compact and simpler form can be obtained:*

$$-\frac{25}{128}k - \frac{(i + 888) \bmod 1024}{1024} + \frac{(i + 200k + 200) \bmod 1024}{1024} + \frac{2539}{64}. \tag{4.47}$$

4.4 Enumerating Polytopes using Barvinok's Decomposition

Here, an analytical method for generating parameterized polytopes is given. This method is an alternative to the interpolation method described in section 4.3.2. The analytical method presented below alleviates the three limitations of the interpolation method: (1) Counting non-parameterized polytopes takes time proportional to the number of vertices, not to the volume of the polytope; (2) degenerate domains do not occur; (3) periodic numbers can be represented using modulo-operations, which is potentially much more compact than the array-representation used in the interpolation method. This method has been developed in close collaboration with Sven Verdoolaege from Katholieke Universiteit Leuven and Rachid Seghir and Vincent Loechner from Université Louis Pasteur Strasbourg.

4.4.1 Non-Parameterized Polytope Counting

The basic idea behind Barvinok's algorithm [13, 34, 57] is to consider the *generating function* of the integer points in a polytope P. This generating function is a formal power series with a term for each integer point in P, i.e.,

$$f(P;\mathbf{x}) = \sum_{\alpha \in P \cap \mathbb{Z}^d} \mathbf{x}^{\alpha}, \tag{4.48}$$

with $\mathbf{x}^{\alpha} = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_d^{\alpha_d}$. Evaluating this function at $\mathbf{x} = \mathbf{1}$ yields the number of terms, which equals the desired number of points. The generating function is obviously not constructed by enumerating all the integer points in P, but rather as a signed sum of short[1] rational functions that can be derived from the description of P.

Example 19. *Consider the polytope P shown in Figure 4.19: $P = \{\mathbf{x} |\ x_1 \geq 0 \ \wedge \ x_2 \geq 0 \ \wedge \ x_1 + x_2 \leq 2\ \}$. Its generating function is $f(P;\mathbf{x}) = 1 + x_1 + x_1^2 + x_2 + x_1 x_2 + x_2^2$. Barvinok's algorithm, however, will produce*

[1] Barvinok uses the term "short rational function" for a rational generating function that describes the integer points in a polytope, for which the size of the rational function is only polynomially large in function of the size of matrix A that defines that polytope (see definition 12).

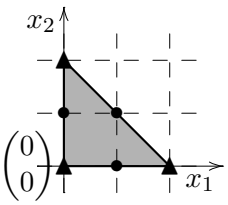


Figure 4.19: Example polytope for which the generating function is $1 + x_1 + x_1^2 + x_2 + x_1x_2 + x_2^2$.

this function in the following form:

$$\frac{x_2^2}{(1-x_2^{-1})(1-x_1x_2^{-1})} + \frac{x_1^2}{(1-x_1^{-1})(1-x_1^{-1}x_2)} + \frac{1}{(1-x_1)(1-x_2)}, \tag{4.49}$$

When evaluating the polynomial at $\mathbf{x} = \mathbf{1}$, the number of integer points in P is found: $f(P;\mathbf{1}) = 6$.

It has been shown by Barvinok et al. [13] that the generating function of a polyhedron P equals the sum of the generating functions of the supporting cones of its vertices, translated to the corresponding vertex.

Example 20. *The generating function of polytope P in figure 4.19 is the sum of the generating functions of the supporting cones at the vertices $\binom{0}{0}$, $\binom{2}{0}$ and $\binom{0}{2}$. The generating functions of the supporting cones are*

$$f(\text{cone}(P, \tbinom{0}{0}); \mathbf{x}) = \frac{1}{(1-x_1)(1-x_2)} \tag{4.50}$$

$$f(\text{cone}(P, \tbinom{2}{0}); \mathbf{x}) = \frac{1}{(1-x_1^{-1})(1-x_1^{-1}x_2)} \tag{4.51}$$

$$f(\text{cone}(P, \tbinom{0}{2}); \mathbf{x}) = \frac{1}{(1-x_2^{-1})(1-x_1x_2^{-1})} \tag{4.52}$$

The generating functions of the supporting cones, translated to the correspond-

ing vertex, are:

$$f(\begin{pmatrix}0\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\0\end{pmatrix}); \mathbf{x}) = \frac{1}{(1-x_1)(1-x_2)} \tag{4.53}$$

$$f(\begin{pmatrix}2\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}2\\0\end{pmatrix}); \mathbf{x}) = \frac{x_1^2}{(1-x_1^{-1})(1-x_1^{-1}x_2)} \tag{4.54}$$

$$f(\begin{pmatrix}0\\2\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\2\end{pmatrix}); \mathbf{x}) = \frac{x_2^2}{(1-x_2^{-1})(1-x_1x_2^{-1})} \tag{4.55}$$

The generating function of the polytope is the sum of these functions, as shown in equation 4.49.

To construct the generating function of a supporting cone, the cone is first decomposed into a set of *unimodular cones*, using Barvinok's decomposition method [14].

Barvinok proposed to decompose this cone into a signed "sum" of unimodular cones $\{(\epsilon_i, K_i)\} = \mathcal{B}(K)$, with $\epsilon_i \in \{-1, 1\}$ the sign corresponding to unimodular cone K_i. Here, "sum" means that the generating function of K is the signed sum of the generating functions of the unimodular cones. It can be shown [13] that a simple explicit formula exists for the generating function of a unimodular cone:

$$f(K_i; \mathbf{x}) = \prod_{j=1}^{k} \frac{1}{(1-\mathbf{x}^{\mathbf{u}_j^i})}, \tag{4.56}$$

with $\mathbf{u}_j^i$ the generators of K_i. A key feature of Barvinok's decomposition algorithm is that it takes polynomial time for fixed dimensions.

Example 21. *The generators for the vertices of the polytope in figure 4.19 are the following:*

$$\textit{generators of } \text{cone}(P, \begin{pmatrix}0\\0\end{pmatrix}) = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \tag{4.57}$$

$$\textit{generators of } \text{cone}(P, \begin{pmatrix}2\\0\end{pmatrix}) = \begin{pmatrix}-1 & -1\\0 & 1\end{pmatrix} \tag{4.58}$$

$$\textit{generators of } \text{cone}(P, \begin{pmatrix}0\\2\end{pmatrix}) = \begin{pmatrix}0 & 1\\-1 & -1\end{pmatrix} \tag{4.59}$$

For all these vertices, the supporting cone is a unimodular cone, so they don't need to be decomposed to write down the corresponding generating function. The generating functions of these cones are shown in equations (4.50–4.52).

To obtain the final generating function, the generating functions corresponding to the unimodular cones K_i need to be translated to the vertex $\mathbf{v}$. If $\mathbf{v}$ is an integer point, then $\mathbf{v}$ simply needs to be added to all of the exponents in the generating function, which corresponds to a multiplication by $\mathbf{x}^{\mathbf{v}}$. If $\mathbf{v}$ is not integer, however, then another point $\mathbf{v}' = E(\mathbf{v}, K_i)$ must be found such that $\mathbf{x}^{\mathbf{v}'} f(K_i; \mathbf{x})$ generates $K_i + \mathbf{v}$. Since K_i is unimodular, this point exists and is uniquely defined as the smallest integer linear combination of the generators of K_i that lies inside $K_i + \mathbf{v}$ [57]. I.e.,

$$E(\mathbf{v}, K_i) = \sum_j \lceil \lambda_j \rceil \mathbf{u}_j^i, \tag{4.60}$$

where λ is the rational solution to $\mathbf{v} = \sum_j \lambda_j \mathbf{u}_j^i$ and $\lceil . \rceil$ is the upper integer part. Note that if $\mathbf{v}$ is integer, then $E(\mathbf{v}, K_i) = \mathbf{v}$. The final generating function is then

$$f(P; \mathbf{x}) = \sum_{\mathbf{v} \in \mathcal{V}(P)} \sum_{i=1}^{|\mathcal{B}(K_{\mathbf{v}})|} \epsilon_i \frac{\mathbf{x}^{E(\mathbf{v}, K_i)}}{\prod_{j=1}^{d} (1 - \mathbf{x}^{\mathbf{u}_j^i})}. \tag{4.61}$$

Example 22. *Figure 4.20 shows the supporting cone of vertex $\mathbf{v}_1$ which is indicated by thick black lines. A possible signed unimodular decomposition for $\mathrm{cone}(P, \mathbf{v}_1)$ is the pair $\{ (+1, K_1), (+1, K_2) \}$. The generators for K_1 are $\mathbf{u}_1^1$ and $\mathbf{u}_2^1$, while the generators for K_2 are $\mathbf{u}_1^2$ and $\mathbf{u}_2^2$. Let $\mathbf{v}_1' = E(\mathbf{v}_1, K_1)$ and $\mathbf{v}_1'' = E(\mathbf{v}_1, K_2)$. Since both signs are positive, the generating function of the cone is*

$$f(\mathrm{cone}(P, \mathbf{v}_1) + \mathbf{v}_1; \mathbf{x}) = f(\mathbf{v}_1' + K_1; \mathbf{x}) + f(\mathbf{v}_1'' + K_2; \mathbf{x}). \tag{4.62}$$

The integer points in $\mathbf{v}_1' + K_1$ are indicated by $\boxplus$, whereas the integer points in $\mathbf{v}_1' + K_1$ are indicated by $\oplus$.

Since $\mathbf{u}_1^1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $\mathbf{u}_2^1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$, $\mathbf{u}_1^2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\mathbf{u}_2^2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $E(\mathbf{v}_1, K_1) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, $E(\mathbf{v}_1, K_2) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, *the generating function for this cone is*

$$\frac{x_2}{(1 - x_1)(1 - x_1 x_2^{-1})} + \frac{x_1 x_2}{(1 - x_1 x_2)(1 - x_1)} \tag{4.63}$$

In order to find the number of integer points in a polytope, the corresponding generating function should be evaluated at $\mathbf{x} = \mathbf{1}$. However, for each term in (4.61) $\mathbf{x} = \mathbf{1}$ results in a division by zero. Therefore, $\lim_{\mathbf{x} \to \mathbf{1}} f(P; \mathbf{x})$ needs to be calculated. Below, a sketch is given

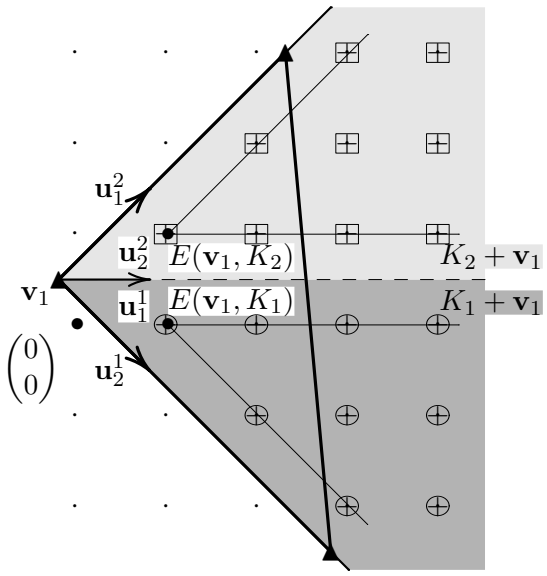


Figure 4.20: Barvinok's decomposition of $\mathrm{cone}(P, \mathbf{v_1})$ into two unimodular cones K_1 and K_2 (indicated by two different shades of grey).

how this can be done systematically, as described by De Loera [57]. Each term in (4.61) can be rewritten (through a suitable variable substitution) as

$$\epsilon_i' \frac{N(s)}{D'(s)} = \epsilon_i' \frac{1}{s^k} \frac{(s+1)^{\langle \mu, E(\mathbf{v}, K_i)\rangle + c}}{D(s)}, \tag{4.64}$$

with $\mathbf{x} = (s+1)^\mu$, $\langle \mu, E(\mathbf{v}, K_i)\rangle$ the inner product of μ and $E(\mathbf{v}, K_i)$, $D(s)$ a polynomial with integer coefficients, independent of $\mathbf{v}$, μ some integer vector and c some integer constant. Evaluating $\lim_{\mathbf{x}\to\mathbf{1}} f(P;\mathbf{x})$ is equivalent to summing $\lim_{s\to 0} \frac{1}{s^k}\frac{N(s)}{D(s)}$. This in turn can be accomplished by computing the coefficient of s^k in the Taylor expansion around 0 of $\frac{N(s)}{D(s)}$ [57]. The Taylor expansion around 0 of $g(s) = \frac{N(s)}{D(s)}$ is given by

$$g(s) = \frac{N(s)}{D(s)} = g(0) + g'(0)s + \frac{g''(0)}{2!}s^2 + \frac{g^{(3)}(0)}{3!}(0)s^3 + \frac{g^{(4)}(0)}{4!}(0)s^4 + \cdots \tag{4.65}$$

Let $N(s) = a_0 + a_1 s + a_2 s^2 + \cdots$ and $D(s) = b_0 + b_1 s + b_2 s^2 + \cdots$. Then the coefficients c_i in the Taylor expansion $\frac{N(s)}{D(s)} = c_0 + c_1 s + c_2 s^2 + \cdots$

are given by the following recurrence relation [57]:

$$\begin{aligned} c_0 &= \frac{a_0}{b_0}, \\ c_i &= \frac{1}{b_0}(a_i - b_1 c_{i-1} - b_2 c_{i-2} - \cdots - b_k c_0) \end{aligned} \tag{4.66}$$

The signed sum of the coefficients c_k of s^k in each of the terms then yields the desired number of points in the polytope.

Example 23. *Consider the triangle polytope in figure 4.19. The generating functions for the cones are given in equations (4.53–4.55). For this example, the generating function is evaluated in the following steps.*

1. *A vector λ is chosen, so that the inner products of λ and the generators of the cones are different from zero. In this example $\lambda = (1, -1)$, and x_i is substituted with t^{λ_i}, resulting in the generating functions:*

$$f(\begin{pmatrix}0\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\0\end{pmatrix}); \mathbf{x}) = \frac{1}{(1-t)(1-t^{-1})} \tag{4.67}$$

$$f(\begin{pmatrix}2\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}2\\0\end{pmatrix}); \mathbf{x}) = \frac{t^2}{(1-t^{-1})(1-t^{-2})} \tag{4.68}$$

$$f(\begin{pmatrix}0\\2\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\2\end{pmatrix}); \mathbf{x}) = \frac{t^{-2}}{(1-t)(1-t^2)} \tag{4.69}$$

2. *Next, the negative exponents in the denominators are eliminated, by simple algebraic rewriting:*

$$f(\begin{pmatrix}0\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\0\end{pmatrix}); \mathbf{x}) = \frac{-t}{(1-t)(1-t)} \tag{4.70}$$

$$f(\begin{pmatrix}2\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}2\\0\end{pmatrix}); \mathbf{x}) = \frac{t^5}{(1-t)(1-t^2)} \tag{4.71}$$

$$f(\begin{pmatrix}0\\2\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\2\end{pmatrix}); \mathbf{x}) = \frac{t^{-2}}{(1-t)(1-t^2)} \tag{4.72}$$

3. *t is substituted with $s + 1$, and the resulting formula is simplified to the*

form $\frac{(s+1)^q}{s^k D(s)}$, where k is the number of generators of the cone:

$$f(\begin{pmatrix}0\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\0\end{pmatrix}); \mathbf{x}) = \frac{-s-1}{s^2} \tag{4.73}$$

$$f(\begin{pmatrix}2\\0\end{pmatrix} + \text{cone}(P, \begin{pmatrix}2\\0\end{pmatrix}); \mathbf{x}) = \frac{s^5+5s^4+10s^3+10s^2+5s+1}{s^2(s+2)} \tag{4.74}$$

$$f(\begin{pmatrix}0\\2\end{pmatrix} + \text{cone}(P, \begin{pmatrix}0\\2\end{pmatrix}); \mathbf{x}) = \frac{1}{s^2(s+2)(s+1)^2} \tag{4.75}$$

4. *The coefficient of the Taylor expansions of $\frac{(s+1)^q}{D(s)}$ around ($s = 0$) is what we are looking for. The Taylor expansions have the following form:*

$$\frac{-s-1}{1} = -1 - s \tag{4.76}$$

$$\frac{s^5+5s^4+10s^3+10s^2+5s+1}{(s+2)} = \frac{1}{2} + \frac{9}{4}s + \frac{31}{8}s^2 + \frac{49}{16}s^3 + \cdots \tag{4.77}$$

$$\frac{1}{(s+2)(s+1)^2} = \frac{1}{2} - \frac{5}{4}s + \frac{17}{8}s^2 - \frac{49}{16}s^3 + \cdots \tag{4.78}$$

The coefficients of the terms s^2 in these Taylor expansions are 0, $\frac{31}{8}$ and $\frac{17}{8}$, respectively. Their sum is 6, which is the number of integer points in the polytope.

4.4.2 Parameterized Polytope Counting

Algorithm 1 shows the extension of Barvinok's method to parameterized polytopes. The main idea behind the generalization is to consider Loechner and Wilde's decomposition of the parameter space in validity domains (see section 4.1.2) and to apply Barvinok's algorithm to the fixed set of parameterized vertices that belong to each validity domain. Thus, one parameterized generating function is computed for each of the validity domains.

Similarly to the non-parameterized case, the generating function for the parameterized polytope $P_{\mathbf{p}}$ in validity domain D is the parameter-

Algorithm 1 Parameterized Barvinok

1. For each vertex $\mathbf{v}_i(\mathbf{p}) \in \mathcal{V}(P)$
   (a) Determine supporting cone $\mathrm{cone}(P, \mathbf{v}_i(\mathbf{p}))$
   (b) Let $K = \mathrm{cone}(P, \mathbf{v}_i(\mathbf{p}))$
   (c) Let $\{(\epsilon_j, K_j)\} = \mathcal{B}(K)$
   (d) For each K_j: Determine $f(K_j; \mathbf{x})$
   (e) $f(\mathrm{cone}(P, \mathbf{v}_i(\mathbf{p})) + \mathbf{v}_i(\mathbf{p}); \mathbf{x}) = \sum_j \epsilon_j \mathbf{x}^{E(\mathbf{v}_i(\mathbf{p}), K_j)} f(K_j; \mathbf{x})$
2. For each validity domain D_k of P
   (a) $f_{D_k}(P; \mathbf{x}) = \sum_{\mathbf{v}_i \in \mathcal{V}_{D_k}(P)} f(\mathrm{cone}(P, \mathbf{v}_i(\mathbf{p})) + \mathbf{v}_i(\mathbf{p}); \mathbf{x})$
   (b) evaluate $f_{D_k}(P; \mathbf{1})$

ized version of equation (4.61):

$$f_D(P_{\mathbf{p}}; \mathbf{x}) = \sum_{\mathbf{v}(\mathbf{p}) \in \mathcal{V}_D(P_{\mathbf{p}})} \sum_{i=1}^{|\mathcal{B}(K_{\mathbf{v}})|} \epsilon_i \frac{\mathbf{x}^{E(\mathbf{v}(\mathbf{p}), K_i)}}{\prod_{j=1}^{d} (1 - \mathbf{x}^{\mathbf{u}_j^i})}, \tag{4.79}$$

with $\epsilon_i \in \{-1, 1\}$ and $\mathbf{v}(\mathbf{p})$ a parameterized vertex of the polytope $P_{\mathbf{p}}$. Each coordinate of $\mathbf{v}(\mathbf{p})$ is an affine function of the parameters. K_i is the ith unimodular cone in the signed unimodular decomposition of cone $K_{\mathbf{v}(\mathbf{p})}$ The correctness of (4.79) follows from the fact that the generators of K are independent of the parameters, which means that Barvinok's decomposition can be applied without change.

The exponent in the numerators of (4.79), which corresponds to the uniquely defined point inside the translated unimodular cone, is given by the parameterized version of (4.60):

$$E(\mathbf{v}(\mathbf{p}), K_i) = \sum_{j=1}^{d} \lceil \lambda_j(\mathbf{p}) \rceil \mathbf{u}_j^i, \tag{4.80}$$

where the $\lambda_j(\mathbf{p})$s are rational affine functions of the parameters that solve $\mathbf{v}(\mathbf{p}) = \sum_{j=1}^{d} \lambda_j(\mathbf{p}) \mathbf{u}_j^i$.

Let m be an integer constant such that $m\lambda_j(\mathbf{p})$ is an integer affine function, i.e. m is the least common multiple of the denominators in

$\lambda_j(\mathbf{p})$, then [79]

$$\lceil \lambda_j(\mathbf{p}) \rceil = \left\lceil \frac{m\lambda_j(\mathbf{p})}{m} \right\rceil = \lambda_j(\mathbf{p}) + \frac{(-m\lambda_j(\mathbf{p})) \bmod m}{m}. \tag{4.81}$$

The second term on the right is a periodic number, say $U'_j(\mathbf{p})$, with period at most m [123]. As explained in section 4.3.1, this periodic number can be represented by an array. The arrays are computed by evaluating the modulo expression in (4.81) for a set of fixed parameter values. Note that, unlike it was the case with interpolation, the values for the parameters need not be restricted to the validity domain since the expression in (4.81) is valid for all values of $\mathbf{p}$. This solves the problem of the degenerate domains. Substituting the value of $\lceil \lambda_j(\mathbf{p}) \rceil$ in (4.80) results in

$$E(\mathbf{v}(\mathbf{p}), K_i) = \sum_{j=1}^{d} \lambda_j(\mathbf{p}) u_j^i + \sum_{j=1}^{d} U'_j(\mathbf{p}) u_j^i = \mathbf{v}(\mathbf{p}) + \mathbf{U}(\mathbf{p}), \tag{4.82}$$

with $\mathbf{U}(\mathbf{p})$ a vector of periodic numbers.

As in the non-parameterized case, the value of $\lim_{\mathbf{x}\to\mathbf{1}} f_D(P_\mathbf{p}; \mathbf{x})$ is obtained by performing the variable substitution proposed by De Loera [57]. The variable substitution is independent of the numerator and hence of the parameters. Substituting (4.82) in (4.64) results in

$$N_\mathbf{p}(s) = (s+1)^{\langle \mu, \mathbf{v}(\mathbf{p}) + \mathbf{U}(\mathbf{p}) \rangle + c} = (s+1)^{\Lambda(\mathbf{p})}, \tag{4.83}$$

where $\Lambda(\mathbf{p})$ is an affine function of the parameters with a constant part that may be a periodic number. The number of points in the polytope equals the sum of the coefficients of s^k in the Taylor expansion of $N_\mathbf{p}(s)/D(s)$.

The coefficients of $N_\mathbf{p}(s)$ up to that of s^k (i.e., those required to compute the coefficient of s^k in the Taylor expansion of $N_\mathbf{p}(s)/D(s)$) are

$$n_i(\mathbf{p}) = \binom{\Lambda(\mathbf{p})}{i} = \frac{\prod_{j=0}^{i-1} (\Lambda(\mathbf{p}) - j)}{i!} \quad \text{for } 0 \le i \le k. \tag{4.84}$$

Each coefficient $n_i(\mathbf{p})$ in the above formula is given by a product of at most d affine functions of the parameters with constant parts that may be periodic numbers. This implies that each of these coefficients is a multivariate polynomial of the parameters in which the coefficients may be periodic numbers and for which the sum of powers in each multivariate monomial is at most d. Since the coefficient of s^k in $N_\mathbf{p}(s)/D(s)$

is a linear combination of these $n_i(\mathbf{p})$ (see equation (4.66)), it conforms to the same property and so does the (signed) sum of all these terms. As a result, $f_D(P_\mathbf{p}; \mathbf{1})$ is an Ehrhart polynomial, as expected:

$$\mathcal{E}_D(P; \mathbf{p}) = f_D(P_\mathbf{p}; \mathbf{1}) = \sum_{0 \leq i_1 + i_2 + \cdots + i_n \leq k} U_\mathbf{i}(\mathbf{p}) \mathbf{p}^\mathbf{i}, \tag{4.85}$$

with the $U_\mathbf{i}(\mathbf{p})$s periodic numbers and d the dimension of $P_\mathbf{p}$. Note that the analytical formulation leads to a tighter general form of the Ehrart polynomials than equation (4.37) in Theorem 5: equation 4.85 shows that there are no terms with a degree higher than k.

The analytical method described above solves the three shortcomings of the interpolation method (see section 4.3.3) in the following ways:

1. This method doesn't need to count the number of integer points in non-parameterized polytopes. Nonetheless, it can be used to count non-parameterized polytopes, without iterating over all the points in the polytope. The complexity of counting non-parameterized polytopes is proportional to the number of vertices, and is independent of the actual number of points in the polytope.

2. Degenerate domains cannot occur, since the entries in the periodic numbers can also be calculated by parameter values outside the validity domains.

3. Long periods can be handled more efficiently by not converting the periodic number in equation (4.81) into an array representation, but keeping modulo operations explicit. An example of the computed Ehrhart polynomial with explicit modulo-operations is given in example 18 on page 142.

Example 24. *As an example, the Ehrhart polynomial of the polytope in figure 4.15 is calculated below. There's a single validity domain with parametric vertices $-2p$ and $\frac{p}{2}$. Vertex $-2p$ has a single unimodular generator 1, while vertex $\frac{p}{2}$ has a single unimodular generator -1. Let's call the unimodular cone of $-2p$ K_0 and the unimodular cone of $\frac{p}{2}$ K_1. The unique integer points $E(-2p, K_0)$ and $E(\frac{p}{2}, K_1)$ are defined by:*

$$E(-2p, K_0) = -2p \times 1 \tag{4.86}$$

$$E(\frac{p}{2}, K_1) = \left\lceil \frac{-p}{2} \right\rceil \times -1 \tag{4.87}$$

Applying equation (4.81), $\lceil \frac{-p}{2} \rceil$ can be rewritten as:

$$\left\lceil \frac{-p}{2} \right\rceil = -\frac{p}{2} + \frac{p \bmod 2}{2} \tag{4.88}$$

Therefore, using equation (4.82),

$$E(\frac{p}{2}, K_1) = \frac{p}{2} - \frac{p \bmod 2}{2} \tag{4.89}$$

The generating function of the polytope is

$$f(P; \mathbf{x}) = \frac{x^{-2p}}{(1-x^1)} + \frac{x^{\frac{p}{2} - \frac{p \bmod 2}{2}}}{(1-x^{-1})} \tag{4.90}$$

after making all the power in the denominators positive, the following formula results:

$$f(P; \mathbf{x}) = \frac{x^{-2p}}{(1-x)} - \frac{x^{1+\frac{p}{2} - \frac{p \bmod 2}{2}}}{(1-x)} \tag{4.91}$$

In order to evaluate this function at $x = 1$, x is first substituted with $(s+1)$:

$$\frac{x^{-2p}}{(1-x)} - \frac{x^{1+\frac{p}{2} - \frac{p \bmod 2}{2}}}{(1-x)} \tag{4.92}$$

$$= \frac{(s+1)^{-2p}}{-s} - \frac{(s+1)^{1+\frac{p}{2} - \frac{p \bmod 2}{2}}}{-s} \tag{4.93}$$

$$= -\frac{1}{s}(s+1)^{-2p} + \frac{1}{s}(s+1)^{1+\frac{p}{2} - \frac{p \bmod 2}{2}} \tag{4.94}$$

Therefore, the count is the signed sum of the coefficient of s^1 of polynomials $(s+1)^{-2p}$ and $(s+1)^{1+\frac{p}{2} - \frac{p \bmod 2}{2}}$. Following equation (4.84), these coefficients are $-2p$ and $1 + \frac{p}{2} - \frac{p \bmod 2}{2}$. The signed sum of these coefficients results in the enumerator for the polytope:

$$\mathcal{E}(P; p) = \frac{5p - p \bmod 2}{2} + 1 \tag{4.95}$$

which is equivalent to the Ehrhart polynomial in equation (4.36).

4.5 Enumerating Parametric Presburger Formula

Computing the reuse distances in a program basically consists of constructing and simplifying the equations (4.13)–(4.21). Equations (4.13)–(4.18) are Presburger formulas, and are simplified using the Omega library [97]. However, for equations (4.19)– (4.21), the number of integer solutions of the Presburger formula $\mathsf{ADS}\,(\mathsf{reuse}\,(r \rightarrow s))$ needs to

be counted. Recently, some practical methods have been proposed to count the number of solutions of Presburger formula *without* parameters [29, 30, 135]. However, they are not applicable here, since the variables in I_r, J_s and $\mathcal{P}$ have to be considered parameters, i.e. the number of solutions needs to be computed in function of the variables in $I_r \cup J_s \cup \mathcal{P}$. In 1994, Pugh has described a set of methods to count the number of parametric solutions in Presburger formula [141]. However, the proposed methods have never been implemented and some steps seem ad hoc and not very general. It is doubtful that Pughs method can enumerate arbitrary parametric Presburger formula.

In this section, a method is described to count the number of solutions of a parametric Presburger formula. The method has been implemented in the PolyAST library, and its complexity and limitations for computing reuse distances is discussed in section 4.6. The Presburger formulas are enumerated in 3 steps:

1. The Presburger formula is converted into disjunctive normal form, using the Omega-library. Each term in the disjunction corresponds to a polytope. However, the different polytopes may overlap, and simply summing the number of integer points in these polytopes would result in potentially counting the same solution multiple times. Therefore, the set of polytopes is converted into a set of disjoint polytopes.

2. Each polytope from the disjoint set is counted, using the method described in section 4.4. For each disjoint polytope, the result is a set of validity domains with corresponding Ehrhart polynomials. In order to combine the counts, a partition of the validity domains of all disjoint polytopes is computed, so that each subset in the partition corresponds to the sum of Ehrhart polynomials corresponding to a fixed number of original validity domains. An example is given in example 26.

3. The result can often be simplified by combining adjacent validity domains which have the same Ehrhart polynomial.

The three steps are discussed in more detail in the sections 4.5.1, 4.5.2 and 4.5.3.

4.5.1 Conversion to Disjoint Disjunctive Normal Form

A pair of overlapping polytopes P_1 and P_2 can be converted into a disjoint set of polytopes by computing $(P_1 \setminus P_2) + (P_1 \cap P_2) + (P_2 \setminus P_1)$. However, the difference between two polytopes is not necessarily a single polytope, and might only be describable by a large number of polytopes. Therefore, the conversion of a set of polytopes into a disjoint set of polytopes might lead to a huge increase in the number of polytopes. This effect is called "splintering" by Pugh [141]. In [141], a set of heuristics is presented to avoid splintering as much as possible, and these are implemented and used in PolyAST to make a set of polytopes disjoint. Each of the resulting disjoint polytopes describes a part of the accessed data set (ADS).

Example 25. *The calculation of the backward reuse distance for the read reference* `A(m,j)` *in the Cholesky factorization, shown in figure 4.13 on page 130, is started with the computation of the direct reuses* $\mathsf{reuse}\,(x \rightarrow \texttt{A(m,j)})$, $\forall x \in \mathcal{R}$, *described by equation (4.13). After simplifying the equations with the Omega-library, one finds that the only reference which generates reuse pairs with* `A(m,j)` *is the write reference* `A(l,j)` *(see figure 4.12):*

$$\mathsf{reuse}\,(\texttt{A(l,j)} \rightarrow \texttt{A(m,j)}) = \\ \left\{(N, j, l, k, j', m') : k = j - 1 \wedge j' = j \wedge m' = l \wedge 2 \leq j < l \leq N\right\} \tag{4.96}$$

The next step is to compute the accessed data set, as described by equation (4.18). Only the data elements of array `A` *need to be considered, since it is the only array in the Cholesky factorization. After conversion into disjoint domains, the accessed data set is described by the following polytopes; where an integer point* (d_0, d_1, j, m, N) *indicates that element* $A(d_0, d_1)$ *is accessed between the access generated by* `A(m,j)` *at iteration* (j, m) *and the previous access to element* $A(m, j)$.

$$\{(d_0, d_1, j, m, N) : 1 \leq d_1 < m < d_0 \leq N\} \tag{4.97}$$
$$\{(d_0, d_1, j, m, N) : d_1 = j \wedge 2 \leq j < d_0 < m \leq N\} \tag{4.98}$$
$$\{(d_0, d_1, j, m, N) : d_1 = j \wedge d_0 = j \wedge 2 \leq j < m \leq N\} \tag{4.99}$$
$$\{(d_0, d_1, j, m, N) : d_0 = j \wedge 1 \leq d_1 < j < m < N\} \tag{4.100}$$
$$\{(d_0, d_1, j, m, N) : d_1 = j \wedge 2 \leq j < m < d_0 \leq N\} \tag{4.101}$$

4.5.2 Enumerating Sets of Disjoint Polytopes

After the conversion into disjoint polytopes, each integer point is present in at most one polytope. Therefore, the number of integer points in the union of polytopes equals the sum of the points in each individual polytope. The number of points in each polytope is counted using the method described in section 4.4.2. As a result, for each polytope the number of integer points is expressed in function of the parameters (i.e. the induction variables and the program parameters).

Example 26. *The number of distinct points (d_0, d_1) in the accessed data sets (4.97)–(4.101), in function of j, m and N indicate the number of array elements in the accessed data set for the backward reuses of reference* `A(m,j)`. *The validity domains and corresponding Ehrhart polynomials for the accessed data sets are as follows:*

polytope	validity domain	Ehrhart polynomial
4.97	$\{(j, m, N) : 2 \leq j < m < N\}$	$(N - m)j + m - N$
4.98	$\{(j, m, N) : 2 \leq j < m - 1 < N\}$	$m - j - 1$
4.99	$\{(j, m, N) : 2 \leq j < m \leq N\}$	1
4.100	$\{(j, m, N) : 2 \leq j < m < N\}$	$j - 1$
4.101	$\{(j, m, N) : 2 \leq j < m < N\}$	$N - m$

For a given iteration (j, m), the backward reuse distance is the number of array elements (d_0, d_1) in the accessed data sets, i.e. the sum of the Ehrhart polynomials for which (j, m) is part of the corresponding validity domain. For example, iteration $(j = 5, m = 6)$, with $N > 6$ is part of the validity domains of (4.97), (4.99), (4.100) and (4.101). Therefore, the backward reuse distance for that iteration is $(N - m)j + m - N + 1 + j - 1 + N - m = (N - m + 1)j = (N - 5)5 = 5N - 25$.

The example above shows that in order to calculate the reuse distance of a given iteration, different Ehrhart polynomials from different validity domains need to be summed together. The underlying reason is that the validity domains overlap. In the PolyAST library, the description of the reuse distances is simplified, by transforming the count so that the validity domains form disjoint subsets of the iteration space. It is desirable that the number of different validity domains is as small as possible. However, the conversion of the overlapping validity domains into disjoint sets might result in splintering (see section 4.5.1). To minimize the number of disjoint domains, the following heuristics are used (this is more formally described in algorithm 2):

Algorithm 2 Constructing disjoint validity domains

Input: a set of validity domains $V = v_1, \ldots, v_n$ and corresponding Ehrhart polynomials $ep_1, \ldots, ep_n$.

Output: an equivalent set of disjoint validity domains with corresponding Ehrhart polynomials.

1: Construct graph $G = (V, E)$. The vertices V of the graph are the validity domains $v_1, \ldots, v_n$. There's an edge e_{ij} in the graph if v_i and v_j overlap.
2: Compute the connected components CC of graph G.
3: **while** $\exists cc_i \in CC$ with at least two vertices **do**
4: Split cc_i in two partitions cc_{p1} and cc_{p2}.
5: **for all** edge e_{ij} such that $v_i \in cc_{p1}$ and $v_j \in cc_{p2}$ **do**
6: numpol(e_{ij}) = number of disjoint polytopes in $(v_i \setminus v_j) \cup (v_j \setminus v_i) \cup (v_i \cap v_j)$.
7: **end for**
8: Pick $e_{i'j'}$ for which numpol($e_{i'j'}$) is minimal.
9: Erase $v_{i'}$ and $v_{j'}$ from graph G.
10: Compute disjoint set of polytopes $v_{n+1}, \ldots, v_m = v_{i'} \setminus v_{j'}$.
11: Compute disjoint set of polytopes $v_{m+1}, \ldots, v_o = v_{j'} \setminus v_{i'}$.
12: Compute disjoint set of polytopes $v_{o+1} = v_{i'} \cap v_{j'}$.
13: The corresponding Ehrhart polynomials $ep_{n+1}, \ldots, ep_{o+1}$ are computed as
$$\begin{cases} ep_{n+1} = \cdots = ep_m = ep_{i'} \\ ep_{m+1} = \cdots = ep_o = ep_{j'} \\ ep_{o+1} = ep_{i'} + p_{j'} \end{cases}$$
14: Insert $v_{n+1}, \ldots, v_m, v_{m+1}, \ldots, v_o, v_{o+1}$ in graph G.
15: Construct edges between old nodes in G and the new nodes $v_{n+1}, \ldots, v_m, v_{m+1}, \ldots, v_o, v_{o+1}$, for each pair that overlaps.
16: Recompute connected components CC.
17: **end while**

1. First, a graph is constructed which has a vertex for each validity domain. An edge is drawn between the graph vertices if the corresponding validity domains overlap.

2. The graph is split into connected components.

3. Each connected component is partitioned into two more or less equal parts, so that the number of edges between the two parts are as small as possible. This step is performed by the heuristic algorithms implemented in the METIS software [94]. The idea is to try to cut the connected component into two disjoint connected components, each only half the size of the original connected component.

4. One of the edges crossing the partitioning is chosen. This corresponds to the overlapping of two validity domains V_1 and V_2, with corresponding Ehrhart polynomials E_1 and E_2. These validity domains are made disjoint by computing

$$V_3 = V_1 \setminus V_2 \text{ with corresponding Ehrhart polynomial } E_1 \tag{4.102}$$

$$V_4 = V_2 \setminus V_1 \text{ with corresponding Ehrhart polynomial } E_2 \tag{4.103}$$

$$V_5 = V_1 \cap V_2 \text{ with corresponding Ehrhart polynomial } E_1 + E_2 \tag{4.104}$$

   V_5 is a single polytope, V_3 and V_4 are sets of disjoint polytopes. The polytopes in V_3, V_4 and V_5 are added to the graph, and V_1 and V_2 are removed.

5. The previous step is repeated until all validity domains in the graph are disjoint.

Example 27. *The validity domains in example 26 are made disjoint in the following steps. In each step, the graph of validity domains is shown. Each graph vertex contains the node number in bold text, and the linear inequalities that define the validity domain on the first line. On the second line, the original validity domains are indicated that completely cover that domain. As a result, the Ehrhart polynomial corresponding to a domain is the sum of the Ehrhart polynomials in the covering original validity domains. The dashed line indicates the partitioning computed by METIS, and the bold line indicates the two validity domains that are made disjoint in the next step.*

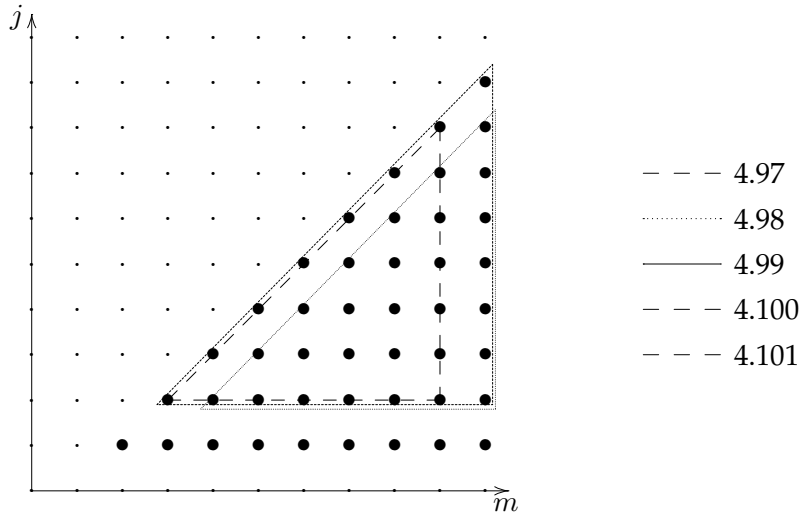
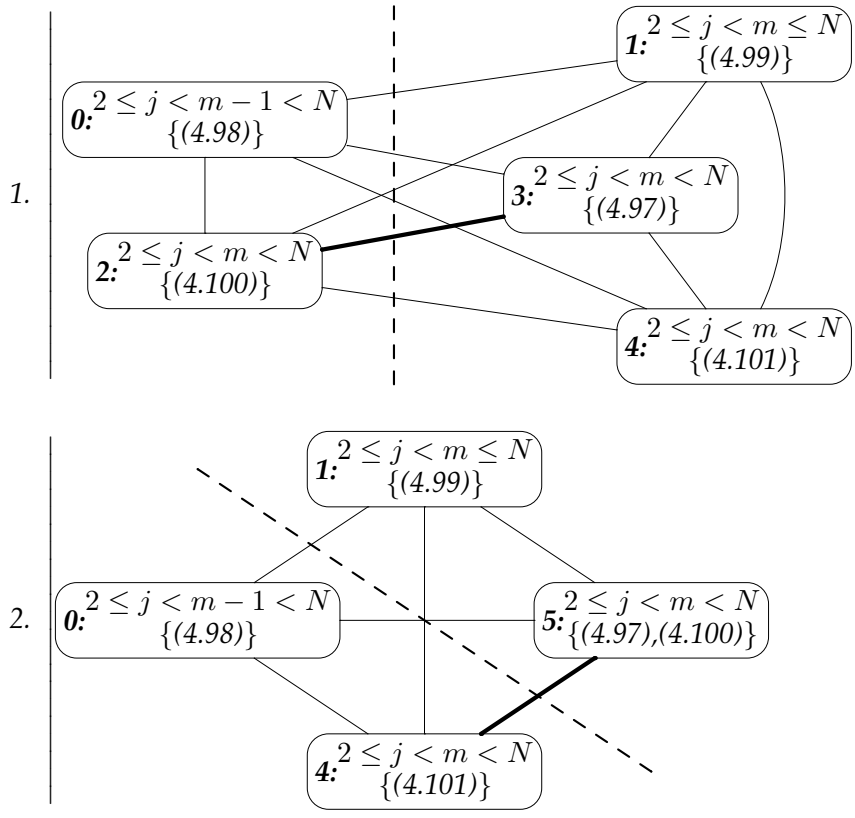
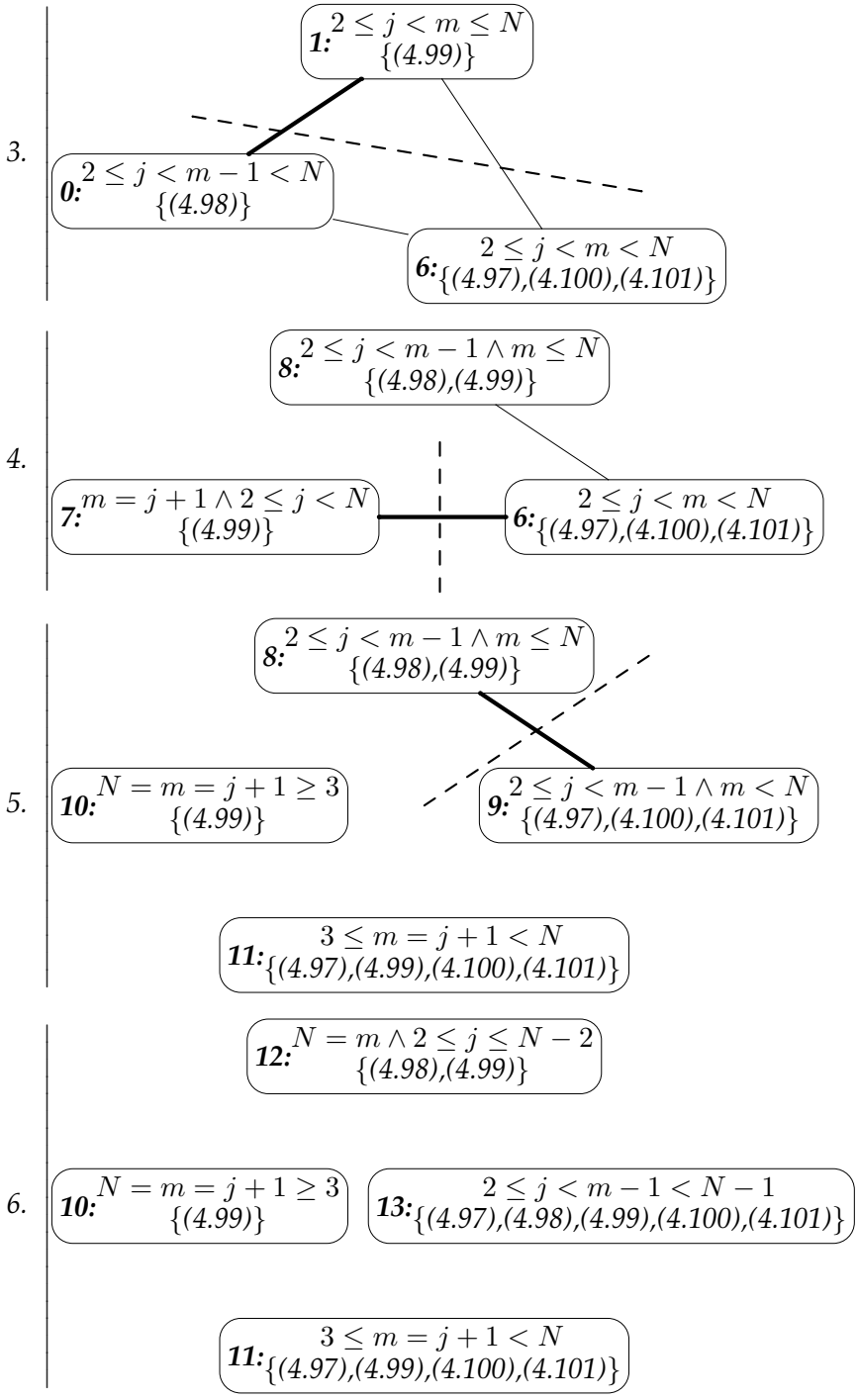


Figure 4.21: Geometrical representation of the overlapping iteration domains 4.97, 4.98, 4.99, 4.100, 4.101.





So, there are 4 disjoint parts in the iteration space, each with their own Ehrhart

polynomial describing the backward reuse distance. The Ehrhart polynomials equal the sums of the Ehrart-polynomials of the corresponding original validity domains:

domain	polynomial
10	1
11	$((N-m)j) + (m-j-1) + (1) + (j-1+N-m)$ $= (N-m+1)j$
12	$(m-j-1) + (1)$ $= (m-j)$
13	$((N-m)j) + (m-N) + (m-j-1) + (1) + (j-1) + (N-m)$ $= (N-m)j + m - 1$

This corresponds to the finite backward reuse distances shown in figure 4.13. The infinite reuse distance in the fifth iteration domain in figure 4.13 is computed by subtracting the domains 10, 11, 12 and 13 from the iteration space. The left-over iterations are iterations which access data for the first time, hence reuse distance ∞.

4.5.3 Simplified Enumerations

The enumeration resulting from algorithm 2 can sometimes be simplified further:

Example 28. *Consider the code in figure 4.22. The backward reuse distance calculation for reference* `A(k)` *results in the following enumeration after applying algorithm 2:*

$$\mathsf{BRD}(\mathtt{A(k)}) = \begin{cases} N-k & \text{if } k = 1 \wedge 2 \leq N \\ k-1 & \text{if } k = N \wedge 2 \leq N \\ N-1 & \text{if } 2 \leq k < N \end{cases} \tag{4.105}$$

In this enumeration, the Ehrhart polynomials from the different domains actually represent the same count $(N-1)$. As a result, the different validity domains can be combined by the following steps.

1. *In the first domain, $k = 1$. Therefore, in the corresponding polynomial, k can be substituted by 1. After this substitution, the polynomial becomes $N-1$. Furthermore, the union of the first domain and the third*

```
do i=1,N
  A(i)=5
enddo
do k=1,N
  A(k)=6
enddo
```

Figure 4.22: Code for which the calculation of $\mathsf{BRD}(\mathtt{A(k)})$ illustrates the simplification of enumerations.

domain is the convex polytope $1 \leq k < N$*. Therefore, the enumerator can be simplified by merging the two domains:*

$$\mathsf{BRD}(\mathtt{A(k)}) = \begin{cases} k-1 & \textit{if } k = N \wedge 2 \leq N \\ N-1 & \textit{if } 1 \leq k < N \end{cases} \tag{4.106}$$

2. *Using a similar reasoning, the polynomial of the first domain equals* $N-1$*, and the union of the two domains form a convex polytope. Therefore, they can be merged:*

$$\mathsf{BRD}(\mathtt{A(k)}) = N-1 \textit{ if } 1 \leq k \leq N \tag{4.107}$$

The algorithm implemented in the PolyAST library which performs the above two simplification steps is more formally described in algorithm 3.

4.6 Experiments

The reuse distance equations described above were implemented in the PolyAST library. This library was developed to easily represent programs in the polyhedral model, and to easily describe program properties such as iteration spaces and lexicographical ordering using polytopes and Presburger formulas (PolyAST is short for Polyhedral Abstract Syntax Tree, indicating that the intermediate format to represent programs also contains information encoded as polyhedra). The Omega-library [97] is used extensively to simplify Presburger formula and to convert them into sets of disjoint polytopes. The Polylib [138, 187] library is used to perform operations on polytopes, such as intersection, union, and Clauss' interpolation method for enumerating [50]. Furthermore, the analytical enumeration method described in section 4.4 has been implemented in a separate library, called the Barvinok

Algorithm 3 Simplification of Enumeration

Input: a set of disjoint validity domains $V = v_1, \ldots, v_n$ and corresponding Ehrhart polynomials $P = p_1, \ldots, p_n$.

Output: a simplified set of disjoint validity domains V' with corresponding Ehrhart polynomials.

```
1:  while ∃v_i, v_j ∈ V so that v_i ∪ v_j is a convex polytope do
2:    if p_i = p_j in domain v_j then
3:      v_{n+1} = v_i ∪ v_j
4:      p_{n+1} = p_i
5:      remove v_i and v_j from V.
6:      add v_{n+1} to V.
7:    else if p_j = p_i in domain v_i then
8:      v_{n+1} = v_i ∪ v_j
9:      p_{n+1} = p_j
10:     remove v_i and v_j from V.
11:     add v_{n+1} to V.
12:   end if
13: end while
```

library [183]. Further polyhedral libraries such as CLoog [15] (which generates code from a polyhedral description of the program) and PIP [66, 67] (that computes maxima and minima of linear functions in a parametric polytope), are also integrated into the PolyAST library. Finally, in the experiments, the PolyAST library uses FPT [58], which is a parallelizing Fortran compiler with C [163] and Java [19, 26] front-ends, to parse programs.

4.6.1 Reuse Distance Calculation

In order to verify the correctness of the equations (4.13)–(4.24), they have been calculated automatically for a number of loop-oriented programs, such as the matrix multiplication, Gauss-Jordan elimination, Cholesky factorization and the tomcatv program from the Specbenchmark. Furthermore, they were applied to a number of artificial loop nests which were constructed specifically to lead to far more irregular reuse distances. The results of the analyses were compared to cache simulation, and were identical in all cases.

The experiments described below were all conducted on 7 programs from the NAS benchmark (`mxm`), SPECFP benchmark (`vpenta`,

program	nr. of references	loop nest depth	parametric loop bounds
vpenta	120	2	no
mxm	5	3	yes
liv18	61	2	no
cholesky	9	3	yes
jacobi	7	2	no
gauss-jordan	15	3	yes
tomcatv	77	3	yes

Table 4.4: Number of references and maximum loop depth of the benchmark programs. The third column indicates whether the loop bounds contain parameters.

`tomcatv`), the Livermore loops (`liv18`), and some often-used loop kernels (`jacobi`, `gauss-jordan`, `cholesky`). The number of references and the depth of the loop nests for these programs is shown in table 4.4.

The execution time of the different steps in the reuse distance calculation is shown in figure 4.23. The time was measured on a 2.66Ghz Pentium4 running Linux. The algorithms were implemented in C++, and the STL classes are used extensively. During the implementation of the different steps, the focus was on correctness and keeping the code as simple as possible. No effort was performed to make the code efficient. Therefore, the running time of the reuse distance calculation can probably be improved substantially by more careful coding. The execution time is broken down into 5 parts:

1. The time needed to simplify the r^2 Presburger formulas resulting from equation (4.13), where r is the number of references in the program. Notice that although sub-equation (4.14d) can give rise to a large Presburger-formula, the time to process and simplify this formula is actually only about 10% of the total reuse distance calculation time.

2. The execution time of the ADS calculation (equation (4.18)) and its conversion into disjoint polytopes.

3. The execution time of algorithm 1, which counts the number of integer points in the disjoint polytopes describing the ADS.

4. The time needed to make the resulting validity domains disjoint,

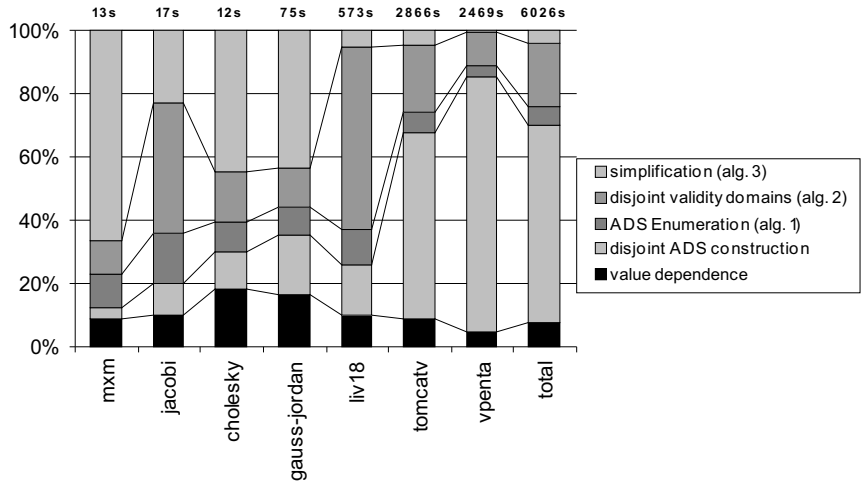


Figure 4.23: Relative computation time of the different steps in BRD calculation. The total execution time is shown at the top of each bar (as measured on a 2.66Ghz Pentium4 processor).

following algorithm 2.

5. The time needed to simplify the result as done by algorithm 3.

The number of generated validity domains, both before and after simplification (see algorithm 3) is shown in table 4.5. The table shows that the simplification results in about 21% fewer different validity domains. On average, the reuse distance of a reference is described by 9 different Ehrhart polynomials, for 9 different parts of the iteration space.

The tables 4.6 and 4.7 show the number of parametric polytopes that need to be counted by algorithm 1, for computing the forward and the backward reuse distance respectively. The tables show that the interpolation method can compute most polytopes, but for a few polytopes degenerate domains result. In contrast, the method based on Barvinok's decomposition finds the solution for all polytopes. Furthermore, it is about 2 times faster than the interpolation method.

During the experiments, it showed that for most references, most of its iterations lay in a single iteration domain. As an example, consider the read reference `A(m,j)` in the Cholesky factorization once more. The majority of the iteration points lay in iteration domain 3 in fig-

program	BRD			FRD		
	domains before simplification	domains after simplification	average domains per reference	domains before simplification	domains after simplification	average domains per reference
vpenta	552	475	3.96	552	474	3.95
mxm	72	60	12.00	72	60	12.00
liv18	821	692	11.34	796	695	11.39
cholesky	87	70	7.78	88	71	7.89
jacobi	116	89	12.71	131	91	13.00
gauss-jordan	187	161	10.73	188	162	10.80
tomcatv	1395	1110	14.42	1377	1092	9.00
total	3230	2657	9.04	3204	2645	9.00

Table 4.5: Total number of iteration domains before and after simplification. The simplification results in reuse distances being described by 21% fewer iteration space domains.

program	nr. of polytopes	degenerate domains	exec. time interpolation	exec. time Barvinok
vpenta	3248	0	131.62s	106.54s
mxm	33	0	2.50s	0.96s
liv18*	2648	3	118.89s*	85.75s
cholesky	38	0	2.50s	1.05s
jacobi*	123	3	5.47s*	3.30s
gauss-jordan	154	0	8.91s	4.27s
tomcatv*	4393	33	336.97s*	148.58s
total	10637	39	606.86s	350.45s

Table 4.6: Number of polytopes that are counted while calculating *forward* reuse distances. For 39 polytopes, the interpolation method cannot find the solution, i.e. degenerate domains occur. The programs for which this happens are indicated by a *. Furthermore, the presented analytical calculation is about 1.73 times faster than the interpolation method.

program	nr. of polytopes	degenerate domains	exec. time interpolation	exec. time Barvinok
vpenta	3248	0	131.35s	87.33s
mxm	33	0	3.61s	0.97s
liv18*	2648	3	116.59s*	70.43s
cholesky	38	0	2.36s	1.00s
jacobi*	123	3	5.29s*	3.17s
gauss-jordan	154	0	9.00s	4.09s
tomcatv*	4393	33	326.50s*	125.94s
total	10637	39	594.70s	292.93s

Table 4.7: Number of polytopes that are counted while calculating *backward* reuse distances. For 39 polytopes, the interpolation method cannot find the solution, i.e. degenerate domains occur. The programs for which this happens are indicated by a *. Furthermore, the presented analytical calculation is about 2 times faster than the interpolation method.

program	nr. of constant dominant polynoms	nr. of linear dominant polynoms	nr. of quadratic dominant polynoms	nr. of polynoms with inner induction var
vpenta	68	52	0	53
mxm	2	1	2	0
liv18	41	20	0	13
cholesky	6	1	2	1
jacobi	5	2	0	0
gauss-jordan	10	4	1	1
tomcatv	39	13	25	17
total	171	93	30	85

Table 4.8: Number of constant, linear and quadratic dominant polynomials. For 85 references, the dominant polynomial depends on the innermost induction variable, i.e. the reuse distance of 85 references changes for consecutive iterations of the innermost loop.

ure 4.13, on page 130. These iteration domains are called *dominant domains*:

Definition 29. *For a reference r, an iteration domain with a corresponding polynomial, describing r's reuse distance, is a* **dominant domain** *if and only if the dimension of that domain is the maximum of all iteration domains for that reference. The Ehrhart polynomial corresponding with a dominant domain is called a* **dominant reuse distance polynomial**.

Example 29. *Iteration domain 3 of the read reference* `A(m,j)` *in the Cholesky factorization has dimension 2, while all other domains have at most dimension 1. Therefore, iteration domain 3 is the only dominant domain of that reference. The corresponding dominant reuse distance polynomial is* $(N-m)j+m-1$.

Of the 294 references in total, only 7 references exhibit 2 dominant domains for their forward reuse distance. All the other references have only one single dominant domain. For 3 of the 7 references with 2 dominant domains, the Ehrart polynomials of the dominant domains were identical. For the 4 other references, the difference between both dominant polynomials was 1. Therefore, the forward reuse distance of the majority of all memory accesses can be described by a single Ehrhart polynomial for almost all references.

program	nr. of formulas	unhandled formulas
vpenta	120^2=14400	17
mxm	5^2=25	4
liv18	61^2=3721	31
cholesky	9^2=81	4
jacobi	7^2=49	5
gauss-jordan	15^2=225	*
tomcatv	77^2=5929	32

Table 4.9: The number of Presburger formulas describing value-based dependences that cannot be processed by the Omega library. The cache line size was assumed to be 4 array elements. The number of formulas is the square of the number of references in the program. For gauss-jordan, the Omega-library crashes on one of the formulas.

The dominant polynomials have a degree at most 2 in the considered programs. Table 4.8 shows the number of constant, linear and quadratic dominant polynomials for each program. It shows that most reuse distances are actually constant, i.e. independent of loop induction variables or program input. This observation coincides with earlier observations in the literature [1], which suggested that only 10% of the memory instructions cause cache misses. However, a substantial part of the references exhibit linear or quadratic reuse distance in function of induction variables and program parameters. For these references, the cache behavior is dependent on the size of the program input, even though the program input might have to be very large before some of these references start to generate cache misses [200].

4.6.2 Taking into account Cache Line Size

When longer cache lines are taken into account (see section 4.2.5), two additional problems arise when solving the equations:

1. The Omega-library doesn't always succeed in converting the Presburger formula (4.13) into a disjunctive normal form(DNF). When the Presburger formula is too complex for the Omega library to handle, it generates a disjunctive normal form, with one additional UNKNOWN constraint, indicating that there are additional disjunctions (polytopes) in the DNF, but the Omega library is unable to generate them [195]. Table 4.9 shows for each program how many Presburger formula resulting from equa-

Program	# polytopes	degenerate domains	>10s interpol.	max. period	matrix sizes
gauss	55	4	10	401	401×400
mxm	398	40	9	800	400×400
jacobi	277	15	8	260	1040×1040
liv18	3436	388	165	4	1024×1024
total	4166	447	192		

Table 4.10: Polytope counting for cache line size=4.

tion (4.13) that need to be converted into disjunctive normal form. It also shows for how many of those, the Omega library fails to generate all polytopes. It shows that for none of the programs, all formulas can be solved.

2. The conversion into disjoint disjunctive normal form takes a long time. Only for the programs gauss-jordan, jacobi, liv18 and mxm, the conversion was performed in less than a day of processing time.

3. The longer line size results in larger periodic behavior, leading to Ehrhart polynomials with larger periods. Therefore, the array representation of periodic numbers becomes large, and the interpolation method needs to interpolate over a large number of parameter points to find all unknown coefficients. However, this can be solved by using the method based on Barvinok's decomposition, and using the modulo-representation as shown in example 18 on page 142. Table 4.10 shows that more than 10% of these polytopes result in degenerate domains for these polytopes. Furthermore, 192 of these polytopes need a large time for counting the number of solutions for specific parameter values.

4.7 Related Work

In recent years, a few different approaches have been proposed to calculate locality and cache behavior at compile time, without profiling. The seminal work by Ghosh on Cache Miss Equations (CME) [75, 76, 77] constructs equations describing cache behavior based on reuse vectors [125]. However, these CME's have a number of limitations. Since

Method	exact	multiple loop nests	probabilistic	models parametric input size	solvable for non-unit cache line size
CME [75]	-(rv)	-	-	-	-
PME [71, 72]	-(rv)	+	+	-	+
Harper [82]	-	+	+	-	+
Vera [181, 182]	-(rv)	+	+	-	+
Chatterjee [44]	+	+	-	-	+
Cascaval [39]	-(rv)	-	-	+	+
RDE(this work)	+	+	-	+	-

Table 4.11: Comparison between different analytical cache equation models. (rv)=equations based on reuse vectors.

they are based on reuse vectors, they only capture the reuses between uniform references, i.e. where the array index expressions differs at most by a constant. Furthermore, in order to find the memory accesses that cause cache misses, each iteration point of each reference needs to be considered individually. Basically, this results in simulating all the memory accesses. However, in contrast to cache simulation where first all previous accesses must be simulated before the cache behavior of the current access can be determined, the CME's allow to calculate an access's cache behavior independent from the surrounding memory accesses. This allows to statistically take a sample of iteration points to probabilistically calculate the cache behavior of references, as proposed by Vera [181, 182]. Other equations based on probabilistic reasoning have also been proposed, e.g. by Fraguela [71, 72] and Harper [82].

The cache equations that come closest to the reuse distance equations are the Presburger formulas presented by Chatterjee [44] and the stack distance computation presented by Caşcaval [39]. In [44], Chatterjee proposes *exact* modelling of cache behavior, based on Presburger arithmetic. However, taking line size into account seems to generate formulas which are too complex to be simplified by the Omega-library, since the equations are only solved for a matrix multiplication code. Furthermore, symbolic loop boundaries and array sizes are not explicitly considered. In contrast, Caşcaval [39] presents a method to compute reuse distance of perfect loop nests with symbolic loop boundaries. However, the computation only applies to uniform array references in perfect loop nests. The reuse distance equations presented in this chapter differ from these proposals in that it is both exact and, when the cache line size is equal to the array elements size (=unit-sized cache lines), it allows to compute reuse distances for a wide range of programs. When the cache line size is larger and the data size of the matrix is not a known constant, the underlying formulas become non-Presburger, and cannot be solved by current tools.

A comparison of the different methods is summarized in table 4.11. Of all the methods, only those based on reuse distance (Caşcaval's and ours) seems to be able to handle symbolic data sizes, albeit only for unit-sized cache lines. Therefore, these methods seem most applicable in compiler optimizations for general purpose computers, where the array sizes are often unknown at compile time.

In contrast to analytically computing reuse distances for different data sizes, Ding [62] and Zhong [200] compute reuse distance distri-

butions with parametric data size for general programs. This is done by profiling reuse distance distributions for a number of fixed data sizes and extrapolating these measurements. In comparison to reuse distance equations, the profiling method applies to general programs. However, it only predicts the number of cache misses for a complete program execution, and it is not able to identify the specific memory accesses that result in cache misses.

Most compiler optimizations for cache behavior and data locality [4] are based on reuse vectors. Therefore, they are mostly limited to optimizing single loop nests. Furthermore, only the distance in the iteration space is known to these optimizations. Most of these optimizations use rough heuristics to estimate the reuse distance from the iteration space distance between reuses. The reuse distance equations allow a finer analysis, and new compiler optimizations can profit from them, e.g. the dynamic cache hint selection presented in chapter 5.

4.8 Summary

This chapter presents reuse distance equations, that describe reuse distances for programs in the polyhedral model. One of the corner stones in solving the equations is an algorithm to count the number of integer points in sets of parameterized polytopes. The best existing method, based on interpolation techniques, has a number of shortcomings. Therefore, an alternative algorithm which has been developed in collaboration with Sven Verdoolaege, Rachid Seghir and Vincent Loechner, is proposed which eliminates the shortcomings of the interpolation method.

The experiments show that the new method is necessary for solving reuse distance equations, since the old interpolation method gives rise to a number of degenerate domains. Furthermore, the new method allows to represent periodic numbers with large periods with modulo-operations instead of large arrays. This allows to drastically reduce the size of representation for Ehrhart polynomials with large periods.

Next to counting single polytopes, algorithms are presented which combine and simplify the enumerators of sets of disjoint polytopes.

The experiments show that the reuse distance equations can be solved in reasonable time when only temporal locality is considered, i.e. with unit-size cache lines. However, when also spatial locality is

taken into account (by considering larger cache line sizes), the equations can only be specified as Presburger formulas when the matrix dimensions are fixed constants. Even then, the resulting formulas become too complex to solve. This problem might be resolved by reformulating the reuse distance equations, so that simpler Presburger formulas are generated. An alternative way would be to extend the methods to simplify Presburger formula. However, it might also be that cache behavior of programs in caches with line size larger than 1 is simply too irregular to be computed analytically and represented by a concise formula, such as a set of Ehrhart polynomials.

The advantage of the presented reuse distance equations over other methods that calculate cache behavior and locality without profiling is that it combines the following properties:

1. it is exact.
2. it allows symbolic loop boundaries and data sizes, when the cache line size equals the array element size.
3. it handles sequences of imperfect loop nests.

The underlying polyhedral theory gives rise to different domains in the iteration space of a reference. For each iteration domain, one Ehrhart polynomial describes the reuse distance of these memory accesses. For the considered benchmarks, each reference has on average 9 different domains. However, almost all references exhibit a single dominant domain that contains the majority of its iterations. Furthermore, the non-dominant iteration domains are all located at the border of the iteration space, i.e. at the first or last iteration of some enclosing loop. This property can be exploited to peel of first and last loop iterations. In the resulting code, the different references have a single iteration domain describing its reuse distance, which is used in cache hint selection to generate more efficient code, as is later discussed in chapter 5.

Chapter 5

Cache Hint Selection

On every cache miss, the replacement policy chooses a cache line to be evicted from the cache. Shortly after caches had been introduced, it became known that the optimal replacement policy chooses the cache line that will be accessed furthest in the future [16]. However, replacement policies are implemented in the cache hardware and have no information about future memory accesses, making the optimal policy impossible to implement. Most implementable replacement policies keep some information about past accesses to predict the line which is most likely not to be accessed for the longest time. For example, the LRU replacement policy assumes that the cache lines that are accessed furthest in the past are also the lines that will be accessed furthest in the future. In other words, it assumes that the "backward" and the "forward" locality are symmetric.

In contrast to the cache hardware, the compiler may be able to deduce information about future memory accesses. In EPIC architectures, target cache hints have been introduced which allow to communicate this information to the cache hardware, where the replacement decision can take this information into account. In this chapter, it is described how the reuse distance metric is used in the compiler to generate cache hints. Next to target hints, also source hints are generated, which are used in the instruction scheduler to make a better-informed estimate of the latency of load instructions.

5.1 Cache Hints in EPIC Architectures

Recently, steering cache replacement policy from software has been enabled by cache hints, which have emerged in EPIC (=Explicitly Parallel

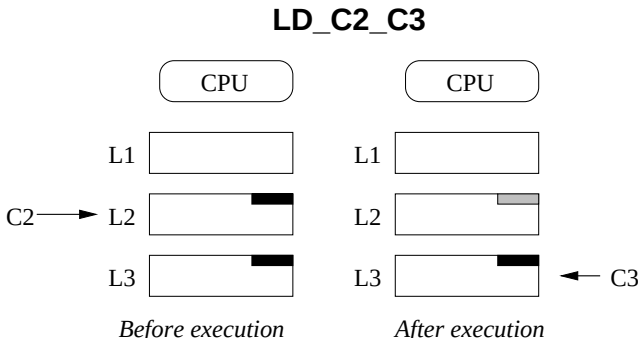


Figure 5.1: Example of the effect of the cache hints in the load instruction `LD_C2_C3`. The source cache specifier `C2` in the instruction suggests that the data resides in the L2-cache. The target cache specifier `C3` indicates that the data should be stored no closer than the L3-cache. As a consequence, the data becomes the first candidate for replacement in the L2-cache.

Instruction Computing) [155] architectures. This is in line with the central idea of the EPIC paradigm: the compiler is responsible for deciding when instructions are issued and which processor resources are used. In contrast to superscalar processors, the compiler specifies explicitly which instructions to execute in parallel, how to predict branches and where to place data in the cache hierarchy. In EPIC instruction sets, cache hints provide a means to the compiler to communicate its decisions about cache placement and replacement to the processor. The major EPIC architectures, the HPL-PD [95] research architecture and the IA-64 [86] architecture, both provide cache hints with similar semantics.

5.1.1 Cache Hints in the HPL-PD Architecture

In the HPL-PD research architecture, cache hints are annotations to regular memory instructions, and exist in two varieties: the source and the target hints. The first kind, the *source cache specifier*, indicates at which cache level the accessed data is likely to be found. The second kind, the *target cache specifier*, indicates at which cache level the data is kept after the instruction is executed. An example is given in figure 5.1, where the effect of the load instruction `LD_C2_C3` is shown.

The *source cache specifiers* are used by the instruction scheduler to know the estimated data access latency. Without these specifiers, the compiler assumes that all memory instructions hit in the L1 cache. Us-

ing the source cache specifier, the compiler is able to better estimate the true memory latency of instructions. When the memory access has a longer latency, the compiler tries to hide that latency by scheduling more instructions explicitly in parallel with the memory operation, resulting in data preloading (cfr. [2]). The *target cache specifiers* are used by the processor, where they indicate the highest cache level at which the data should be kept. A carefully selected target specifier will maintain the data at a fast cache level, while minimizing the cache pollution at the cache levels where the data won't be retained until its next use.

5.1.2 Cache Hints in the IA-64 Architecture

Source hints need to be communicated to the instruction scheduler, while target hints need to be communicated to the cache hardware. In EPIC-architectures, the instruction scheduler is completely located in the compiler (in contrast to superscalar processor where instruction scheduling is done both by the compiler and the processor).

Since instruction scheduling is completely performed in the compiler, the source hints don't need to be communicated to the processor. Therefore, they are not encoded in an industrial EPIC instruction set, such as the IA-64.

In contrast, target hints are encoded in the IA-64 instruction set, since they carry information that needs to cross the compiler/processor interface. The IA-64 ISA defines the target cache hints `.t1`, `.nt1`, `.nt2` and `.nta`. These hints specify whether there is temporal locality at a given cache level. The IA-64 model assumes that there are two parallel cache hierarchies: one for accesses which exhibit temporal locality, and one which exhibits only spatial locality, or even no locality at all. The semantics of the different cache hints in this model are depicted in figure 5.2. `.t1` specifies that the memory instruction has temporal locality at all cache levels; `.nt1` indicates no temporal locality at cache level 1, `.nt2` means no temporal locality at cache level 1 and 2, and `.nta` means no temporal locality at all. It is assumed that all memory accesses exhibit spatial locality.

Despite the model of separate cache hierarchies for temporal and non-temporal data, the first implementations of the IA-64 architecture, the Itanium1 [162] and Itanium2 [87] processors, implement a cache hierarchy with a single cache per level. Nonetheless, the hints influence the replacement policy. The single cache hierarchy on the Itanium pro-

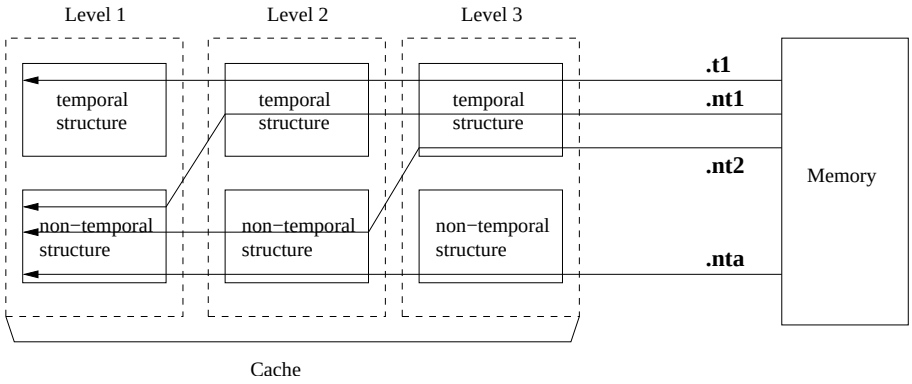


Figure 5.2: The semantics of the target hints in the IA-64 architecture [86]. If the cache hint indicates temporal locality for a given cache level, the data is allocated in the temporal buffer; otherwise it is placed in the non-temporal buffer.

hint	L1		L2		L3	
	alloc	update LRU	alloc	update LRU	alloc	update LRU
`.t1`	√	√	√	√	√	√
`.nt1`			√	√	√	√
`.nt2`			√		√	√
`.nta`			√			

Table 5.1: The effect of the target cache hints on the replacement policy in the Itanium2 processor [87]. alloc=√ means that the data is allocated in that cache level. update LRU=√ means that the LRU bits are updated, i.e. that cache line is considered the most recently accessed line.

cessor is viewed as the temporal cache hierarchy from the IA-64 model. In order to exploit spatial locality for memory accesses where the cache hint indicates no temporal locality, data is always put in a single way in the second cache level. When the cache hint indicates that there is no temporal locality, the corresponding LRU bits are not updated. As a result, for data without temporal locality, the complete memory line is fetched into the L2-cache, where it stays shortly at the bottom of the LRU-stack. It is hoped that the short stay is long enough to exploit the spatial locality of the access. The detailed effect of the different hints are shown in table 5.1.

5.1.3 Static and Dynamic Hints

Both the HPL-PD and the IA-64 architectures specify *fixed* cache hints per instruction. A single memory instruction is typically executed multiple times (e.g. in a loop), and each execution results in a memory access. The memory accesses originating from a single instruction might exhibit different amounts of locality and therefore require different cache hints. When a single fixed cache hint is selected for all the memory accesses generated by a single instruction, it is called a *static hint*. When a hint is tailored towards the locality of the individual accesses, it is called a *dynamic hint*, since these hints are dynamically computed at run-time. In section 5.2, a compile-time generation of static hints is proposed, based on profiled reuse distance distributions. Dynamic hint computation based on reuse distance equations, and the extensions to the ISA it requires, is discussed in section 5.3.

5.2 Static Hint Selection

Traditional software cache optimizations, such as loop tiling [21] reduce the number of cache misses by reducing the reuse distance of the accesses. However, due to data and control dependences, for many programs, it is not possible to legally perform these program transformations. Here, we exploit the possibility to adapt the instruction scheduling and the cache replacement policy through cache hints when the reuse distance is larger than the cache size.

5.2.1 Cache Hint Selection for a Memory Access

The source cache hint should indicate the highest cache level where the data can be found at the time the memory access occurs. If a faster cache level is indicated, the compiler would assume a latency that is smaller than the true latency and it wouldn't try to hide the full latency of the load. If a slower cache level is indicated, the assumed latency would be too large. As a consequence, the compiler would generate a suboptimal schedule, because the target register of the memory instruction will be kept live longer than necessary. This increases register pressure and might lead to more register spill/fill-code. Theorem 1 on page 93 specifies that the backward reuse distance indicates the minimal cache size which is needed for the access to be a cache hit in a fully associative

cache. As described in chapter 3, it also indicates the minimal cache size for the access to be a hit for lower-associative caches with high probability. Therefore, the source cache hints are selected using the following rule:

Source Cache Hint Selection Strategy. *The fastest cache level where data resides at the start of the memory access is the smallest cache level that is larger than the backward reuse distance. The source cache hint is chosen to indicate that cache level.*

The target cache hints should indicate the smallest cache level where the data will be retained until its next use, i.e. the fastest cache level l where the locality of the data will be exploited. If a larger cache level is selected, the locality would not be exploited in the fast level l, leading to an extra miss for that level. If a smaller level is selected, the data is fetched into a level where the locality cannot be exploited. Even worse, the data pollutes the smaller cache, and can throw out other data with higher locality. According to theorem 1, the forward reuse distance indicates the cache size that is needed for the data to be retained until the next reuse in a fully associative cache. As described in chapter 3, it also indicates the cache size needed to keep the data for lower-associative caches with high probability. Therefore, the target hints are selected by the following rule:

Target Cache Hint Selection Strategy. *The target cache hint which indicates the smallest cache level that is larger than the forward reuse distance is selected.*

Further evidence of the appropriateness of this target hint selection scheme is given in [89], where Jain et al. prove that this cache hint choice is guaranteed to perform equal or better than the LRU replacement policy for a fully associative cache.

5.2.2 Cache Hint Selection for a Memory Instruction

A single memory instruction generates multiple memory accesses when that instruction is executed multiple times (e.g. in loops). The different accesses originating from the same memory instruction may exhibit different locality, requiring different cache hints.

However, it is not possible to specify different cache hints, since the hint is specified on the instruction. As a consequence, all accesses

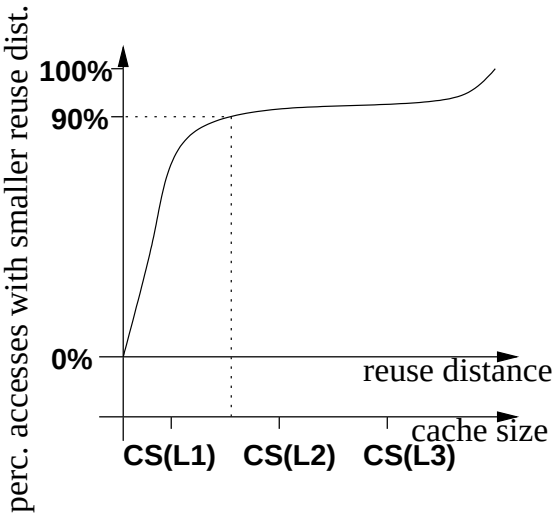


Figure 5.3: A cumulative reuse distance distribution for an instruction is shown and how a threshold value of 90% maps it to cache hint C2. CS(Lx) = cache size for cache level x.

originating from the same instruction share the same cache hint. The following approach is used to obtain a single hint per instruction which is applicable for most accesses generated by the instruction.

First, for every instruction, the cumulative reuse distance distribution of all generated memory accesses is collected. An example of such a distribution is shown in figure 5.3. Based on the distribution, a source cache hint is selected so that for at least x% of the accesses, the data will be found in that cache level. Figure 5.3 shows how to find the cache hint so that for at least 90% of the accesses, the data will be found in the indicated cache level. Similarly, the distribution can be used to select the target cache hint so that for at least y% of the accesses, the data will be retained at that cache level. Clearly, for some distributions, it is impossible to find a cache hint which indicates the correct cache level for the majority of all accesses. Therefore, the hints should be selected so that if they are wrong, they don't incur a high cost, i.e. err on the safe side. The heuristic used to select a single static hint is discussed below.

In table 5.2, the expected cost of too small and too large source and target hints are indicated. It is expected to be less costly to have a source hint which is too large instead of one which is too small. It is costlier to indicate a too small cache level, since the compiler won't try to hide

	indicated level too small	indicated level too large
source cache hint	high cost	low cost
target cache hint	low cost	high cost

Table 5.2: Expected cost of a wrong cache hint.

the full latency of the access. When the hint is too large, the compiler merely tries to schedule more parallel instructions between the access and the next dependent instructions. For a target hint, it is costlier to indicate a cache level too large instead of a level too small. When the hint is too small, the data will be brought in a level where it won't be reused, potentially leading to cache pollution. However, this would happen anyway with an LRU replacement policy. On the other hand, when the hint is too large, the data won't be fetched in all levels where it can be reused, and therefore, unnecessary cache misses will definitely result.

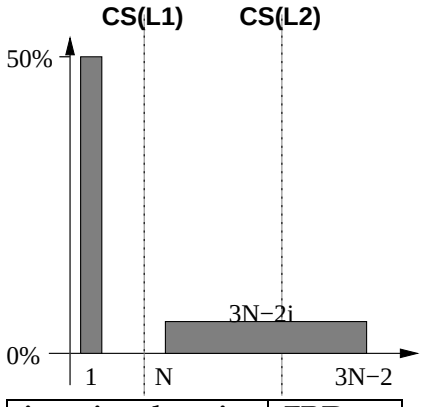
Taking table 5.2 into account, in the experiments, x is chosen to be 90%, while y is chosen to be 10%. In this way, at most 10% of the accesses could have a wrong cache hint which incurs high cost.

The cumulative reuse distance distributions can be measured during a training run of an instrumented version of the program. The implementation of such a profile-based scheme in the Open64-compiler and its application to a number of benchmarks is presented in section 5.4.

5.3 Dynamic Hint Selection

Two major issues arise when using the static cache hint selection scheme described in section 5.2. The first problem is that a cache hint is tied to an instruction, and not to a single memory access. When the reuse distance distribution of the memory instruction shows more than one peak, it is not possible to select the appropriate cache hints for multiple peaks. An example is shown in the bimodal reuse distance distribution of figure 5.4. The second problem is that the locality of the memory accesses generated by an instruction can depend on the input of the program. For example, if the program performs a matrix computation, the size of the matrices can determine the cache level where data will be found. Therefore, the optimal cache hints are also dependent

```
do i=1,N
  do j=1,2
    B(i,j) = A(i)
  enddo
enddo
do j=1,N
  A(j)=0
enddo
```



iteration domain	FRD
$1 \leq i \leq N \wedge j = 1$	1
$1 \leq i \leq N \wedge j = 2$	$3N - 2i$

Figure 5.4: An example of a reuse distance distribution for which it is impossible to statically select good hints for all accesses. On the left hand side, a program is shown, and on the right hand side, the reuse distance distribution of the reference `A(i)` in this program is shown. Half of the accesses generated by `A(i)` exhibit reuse distance 1 demanding for cache hint C1. The other half exhibits reuse distance $3N - 2i$, demanding for a cache hint depending on the values of N and i.

on information that is in general only known at run-time. In figure 5.4, the second peak of memory access `A(i)` is such an example, where the forward reuse distance $3N - 2i$ is dependent on the input size N.

In order to mitigate the above problems, cache hints should be selected at run-time, based on the actual reuse distance of the current access. In this section, the analytically calculated reuse distance (see chapter 4) is used to generate dynamic cache hints. In general, for each reference, the calculation results in a number of domains in the iteration space, where for each domain a polynomial represents the reuse distance. The variables in the polynomial can be the induction variables and the program parameters. For example, for reference `A(i)` in figure 5.4, two different domains in the iteration space are discovered. The first domain exhibits reuse distance 1, while the second domain exhibits reuse distance $3N - 2i$.

```
do i=1,N
 do j=1,2
 if (j.eq.1) then
  FRD_Ai = 1
 if (j.eq.2) then
  FRD_Ai = 3*N-2*i
 B(i,j) = A(i)
 enddo
enddo
do j=1,N
 A(j)=0
enddo
```

(a) exact target hints

```
do i=1,N
 FRD_Ai = 1
 B(i,1) = A(i)
 FRD_Ai = 3*N-2*i
 B(i,2) = A(i)
enddo
do j=1,N
  A(j)=0
enddo
```

(b) after loop peeling

```
FRD_A1 = 1
FRD_A2 = 3*N-2
do i=1,N
 B(i,1) = A(i), frd=FRD_A1
 B(i,2) = A(i), frd=FRD_A2
 FRD_A2 = FRD_A2-2
enddo
do j=1,N
  A(j)=0
enddo
```

(c) after optimization

Figure 5.5: The program in figure 5.4 with dynamic hint calculation for reference A(i).

5.3.1 Code Generation

In a first stage, it is determined in which iteration domain the current iteration lays, by inserting if-tests. For the example code in figure 5.4, the code for computing the forward reuse distance of reference `A(i)` is shown in figure 5.5(a).

The value computed in variable `FRD_Ai` represents the forward reuse distance of reference `A(i)`, which is used to dynamically select the corresponding cache hint. We present two methods to do so. The first method is implementable using the current IA-64 ISA, while the second method requires a small extension to the IA-64 ISA. The extension results in less code duplication and fewer executed instructions at run-time:

1. In order to select the most appropriate cache hint, the load instruction is duplicated with different cache hints. The instruction with the appropriate hint is then selected using predicates. As an example, consider the following code. Assume that the reuse distance value for the given iteration is calculated and stored in register `r10`. The original load instruction loads to register `r5`. `CS1` and `CS2` are the cache sizes of the first level and second level cache. The following IA-64 code executes a single load instruction with the appropriate cache hint, according to the calculated reuse distance:

```
     cmp.lt     p6, p7 = r10, CS1 ;; // FRD < CS1?
(p7) cmp.ge.unc p8, p7 = r10, CS2    // FRD >= CS2?
(p6) ld.t1      r5 = ...          ;;
(p7) ld.nt1     r5 = ...
(p8) ld.nta     r5 = ...
```

   Exactly one of the predicate registers `p6`, `p7`, `p8` will be true after the two `cmp` instructions. `p6` is true if the reuse distance is smaller than the level 1 cache size, `p7` is true if it is between level 1 and level 2 cache size and `p8` is true if it is larger than level 2. A predicate between brackets before an instruction means that the instruction will only be executed if the predicate is true. Consequently, only the load instruction with the proper cache hint will be executed. The instructions between consecutive stop bits `;;` are executed in parallel.

2. A second method requires an architectural extension, allowing more efficient and portable dynamic hints. The memory instructions, such as load, store and prefetch, may have an extra input

register which contains the forward reuse distance of the memory access. An example of such a load instruction is `ld r5=[r6], frd=r7`, where `r7` contains the forward reuse distance. In the encoding of the IA-64 memory instructions, unused bits are available in which the extra input register can be encoded in a backwards compatible way [86].

At compile time, it is not necessary to know the machine dependent cache sizes of the different cache levels, since at run-time the processor will keep the data in the cache levels which are larger than the forward reuse distance. This makes recompilation of the binaries for processors in the IA-64 family with different cache sizes no longer necessary, as far as the target hint selection is concerned. Furthermore, it is easy to generate code without target hints. On IA-64, register `r0` always contains 0. So an instruction like `ld r5=[r6], frd=r0` would specify that the forward reuse distance is 0, leading to the default LRU replacement policy for all cache sizes, since every cache level is larger than 0. Finally, as opposed to the first method, the memory instruction doesn't need to be duplicated for every cache level in the memory hierarchy.

5.3.2 Overhead Reduction

The code and execution time overhead of evaluating the Ehrhart polynomials at run-time can be quite large. The reuse distance of a reference is described by different polynomials for different validity domains. If the reuse distance is described by D domains, before every execution of the reference, D if-tests need to be carried out to calculate which polynomial should be evaluated.

The overhead is reduced by only considering the dominant polynomials. Since most references have only a single dominant domain, most if-tests are eliminated. For the majority of the accesses, the correct reuse distance will be calculated. Furthermore, for the programs used in the experiments section, the non-dominant domains are only located at the borders of the loops, i.e. at the first or last iteration of a loop. So, if more accuracy is wanted, the first or last iteration of a loop can be peeled of. In the peeled iteration, the corresponding polynomial can be used instead of the dominant polynomial. For example, the `j`-loop in figure 5.4 can be peeled (after which it becomes fully unrolled since it

only has 2 iterations). The resulting code is shown in figure 5.5(b).

Furthermore, most reuse distance polynomial evaluations can easily be optimized by standard compiler optimizations such as loop invariant code motion and strength reduction [3]. Many of the computed polynomials are independent of the inner loop variable and can be moved completely out of the inner loop. For the polynomials that are dependent on the induction variable of the inner loop, often strength reduction is able to reduce the polynomial computation to a single addition or subtraction instruction. The result of these optimizations on the running example is shown in figure 5.5(c).

5.4 Experiments

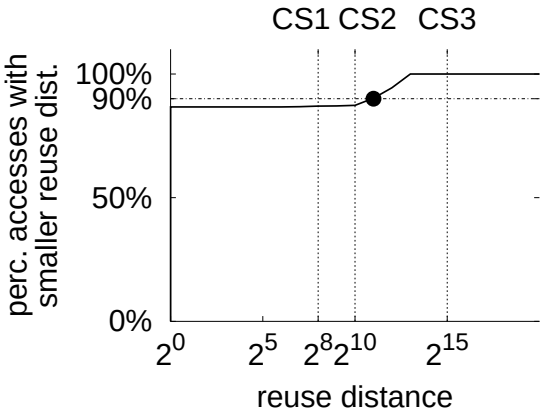
5.4.1 Static Cache Hint Implementation

The static cache hint selection scheme presented in section 5.2 has been implemented in the Open64 compiler for the Itanium [131], which is based on SGI's Pro64 compiler. The reuse distance distributions for the memory instructions are obtained by instrumenting and profiling the program. After profiling, the cumulative reuse distance distribution for every instruction is stored. During the second compilation step, the profile data is read by the compiler and based on the reuse distances, the appropriate source and target cache hints are calculated for each memory operation, in the way described in section 5.2.2 .

The target hints were added to the intermediate representation by representing them as an extra operand to the memory instructions. When the compiler generates assembly code for memory instructions, the corresponding IA-64 cache hint is written out. The source hints were implemented as follows. The compiler back-end has a parametric description of the processor, similar to the machine description system MDES discussed in [145]. The machine description specifies, amongst others, the latency of a given instruction. When the compiler is retargeted to a new processor with different latencies, only the machine description needs to be changed. The source hints were incorporated into the compiler by adding new memory instructions in the machine description table. An example of such an instruction is `ldfd_C3` in figure 5.8. This new instruction has the same characteristics as the existing instruction `ldfd`, except that the machine description indicates the latency of a L3 cache access. In this way, the scheduler automatically

```
DO I1=2,N1-1
   S=S+R(I1,I2,I3)**2
   A=ABS(R(I1,I2,I3))
   IF(A.GT.RNMU)RNMU=A
ENDDO
```

(a) source code from MGRID

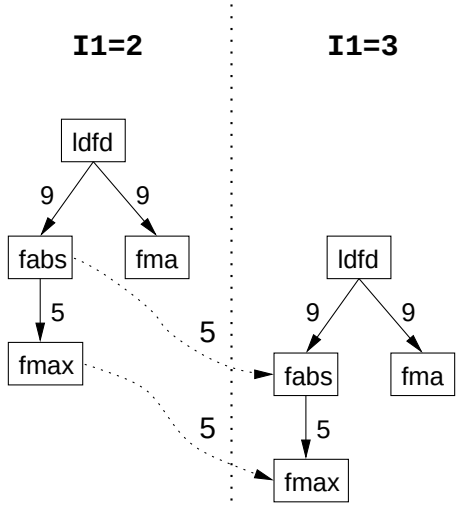


(b) cumulative backward reuse distance distribution

Figure 5.6: Source code from MGRID, and the associated backward reuse distance distribution for reference `R(I1,I2,I3)`.

takes into account the longer latency.

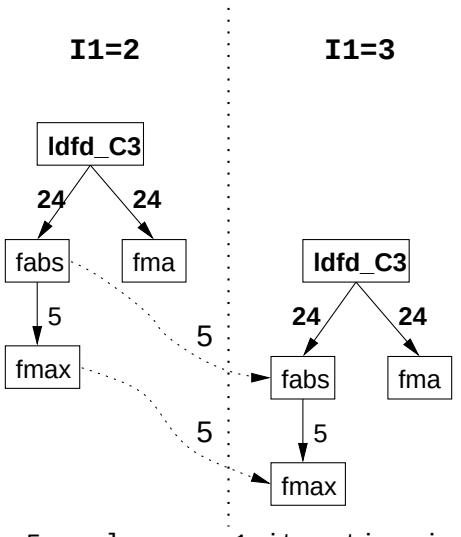
In figures 5.6 and 5.8, an example of the effect of source cache hints for a loop from the mgrid program is shown. The source code is shown in figure 5.6(a). The compiler generates one memory instruction for the loop nest: a load instruction `ldfd` which fetches `R(I1,I2,I3)`. In figure 5.6(b), the cumulative reuse distance distribution of that load instruction is shown. The granularity at which the reuse distance is measured is the memory line, which is 64 bytes on the Itanium, i.e. all accesses to the same consecutive 64 bytes in memory are considered to be accesses to the same data location. Since the loop fetches consecutive data elements from array `R`, and 8 elements fit in a single memory line, 7 out of 8 accesses are to the same memory line as the least recently touched. Therefore 7 out of 8 accesses (=87.5%) have reuse distance 0. The reuse distance of the 8th access, which fetches a new memory line, depends on where that memory line has last been accessed in other




```
//<swps>   5 cycles per 1 iteration in steady state
//<swps>   4 pipeline stages

.Lt_7_17:
    (p18) fabs f34=f38              // [2*II+3]
    (p18) fma.d.s0 f39=f38,f38,f40  // [2*II+3]
 ;;
    (p16) ldfd f36=[r2],8
         // source hint C2 (latency=9) [0*II+4]
    (p19) fmax.s0 f41=f35,f42       // [3*II+4]
          br.ctop.dptk.few .Lt_7_17 // [3*II+4]
 ;;
```

Figure 5.7: Software pipelined schedule without source hints, for the code in figure 5.6. In the dependence graph, the arcs are labelled with the latencies of the instructions.




```
//<swps>   5 cycles per 1 iteration in steady state
//<swps>   7 pipeline stages

.Lt_7_17:
    (p16) ldfd f39=[r35],8
        // source hint C3 (latency=24) [0*II+0]
    (p22) fmax.s0 f47=f38,f48       // [6*II+0]
 ;;
    (p20) fabs f36=f43              // [4*II+4]
    (p20) fma.d.s0 f44=f43,f43,f45  // [4*II+4]
          br.ctop.dptk.few .Lt_7_17 // [6*II+4]
 ;;
```

Figure 5.8: Software pipelined schedule with source hint `C3`, for the code in figure 5.6. The longer latency of the load instruction is hidden by a longer software pipeline (7 stages instead of 4 stages in figure 5.7).

parts of the program. The reuse distance distribution shows that the reuse distance for these accesses are between 2^{10} and 2^{13}. The 90th percentile is at 2^{11} memory lines, hence the source cache hint is chosen to indicate the first larger cache level which contains more than 2048 memory lines, which is the level 3 cache in this case.

In figure 5.7, the original intermediate representation and assembly code for the loop is shown. In figure 5.8, the same code with a source hint is shown. The source cache hint is selected by changing the instruction `ldfd` into `ldfd_C3`. The scheduler knows that the `ldfd_C3` instruction has a latency of 24 cycles. In order to hide this latency, the generated code consists of 7 pipeline stages instead of 4 pipeline stages. Therefore, in the code with hint C3, up to 7 iterations of the original loop execute simultaneously to overlap the longer latency. A drawback of the longer schedule is that more registers are needed (`f36-f48`) than in the original schedule (`f34-f42`). The advantage is that at run-time, because of the correctly scheduled code, the processor won't stall. In the code without source cache hints, the schedule is too tight, and the processor stalls.

The software pipelining scheduler can only generate code when there are enough registers to keep all temporary values, i.e. no spill or fill code needs to be generated. When evaluating the source hint selection, it showed that for a significant number of loops, the register pressure was increased too much by the hint to allow software pipelining. Software pipelining is one of the central optimizations for EPIC architectures like the Itanium, therefore we adapted the source hint selection. When a loop could not be software pipelined because of register pressure, all the hints C4 were replaced with C3. If the loop still required too much registers, the hints C3 were replaced with C2.

5.4.2 Static Hints Experiments

The static selection of cache hints was evaluated on a HP rx4610 multiprocessor, equipped with 4 733MHz Itanium processors. The data cache hierarchy consists of a 16KB L1, 96KB L2 and a 2MB L3 cache. The hardware performance counters of the processor were used to obtain execution time and stall time caused by memory latency.

All compilations were performed at optimization level -O2, the highest level at which instrumentation and profiling is possible in the Open64 compiler. The existing framework doesn't allow to propagate

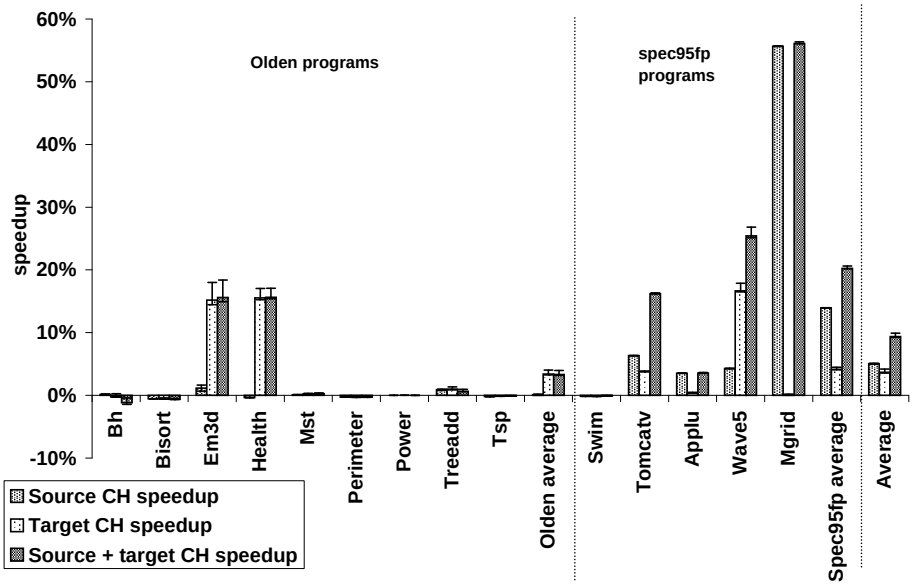


Figure 5.9: Speedup after static source and target hint generation. The error bars indicate the 99% confidence intervals for the speedup, as calculated from 100 program runs.

the profile information through some optimizations phases at level -O3.

The programs were selected from the Olden and the Spec95fp benchmarks. The Olden benchmark contains programs which use dynamic data structures, such as linked lists, trees and quadtrees. The Spec95fp programs are numerical programs with mostly regular array accesses. For Spec95fp, the profiling was done using the train input sets, while the speedup measurements were done with the large input sets. For Olden, no separate input sets are available, and the training input was identical to the input for measuring the speedup. The results of the measurements can be found in figure 5.9.

Execution Time

The figure shows that the programs run 9% faster on average, with a maximum speedup of 56%. In the worst case, a slight performance degradation of 1% is observed. On average, the Olden benchmarks get a 3% speedup from the target hints, but do not profit from the source hints. To take advantage of the source hints, the instruction scheduler

program	source CH speedup with software pipelining	source CH speedup without software pipelining
em3d	1%	1%
treeadd	1%	0%
tomcatv	6%	3%
applu	4%	2%
wave5	4%	0%
mgrid	56%	1%
average	12%	1%

Table 5.3: Speedup after source hint insertion, as compared to a compilation without cache hints. The left column shows the speedup with software pipelining (SWP) and source hints enabled, compared to SWP without source hints. The right column shows the speedup without SWP but with hints, compared to no SWP and no hints.

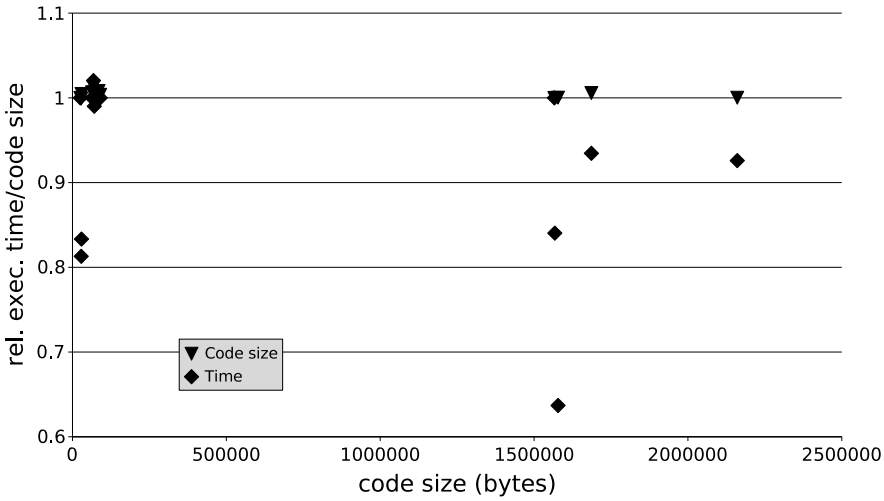


Figure 5.10: Relative execution time and code size for static hints, plotted according to original code size.

must find parallel instructions to fit in between a long latency load and its consuming instructions. In the pointer-based Olden benchmarks, the scheduler finds little parallel instructions, and cannot profit from its better view on the cache behavior. An extra reason of the limited speedup might be that a large part of the working set fits in the L3 cache for these programs.

On the other hand, in the floating point programs, on average a 20% speedup is found, mainly resulting from source hints. The loop parallelism allows the compiler to find parallel instructions, mainly because it allows it to software pipeline (SWP) the loops with long latency loads. Without software pipelining, the compiler doesn't find many parallel instructions to schedule during long latency memory accesses. This can be seen in table 5.3, where the programs are shown which do profit from source hints. When SWP is enabled, adding source hints results in 12% speedup on average. When SWP is disabled, adding source hints result in only 1% speedup on average, because the compiler cannot schedule parallel instructions during long latency memory accesses.

Code Size

The relative code size of the programs with static cache hints is shown in figure 5.10. On the x-axis, the code size of the program without hint selection is shown. The code size of the Olden-programs is much smaller than the code size of the SPEC95 programs.

The target hints have no influence on the code size. For every memory instruction in the IA-64 architecture, 2 bits are reserved to encode the cache hint. Traditionally, compilers just fill in hint `.t1`, whereas our compiler fills in these 2 bits to encode the selected target hint. On the other hand, source hints might change code size, since the compiler creates a different instruction schedule. However, figure 5.10 shows that the change in code size is always less than 1%.

5.4.3 Dynamic Hint Experiments

The method based on profiling sets cache hints per instruction. The analytical calculation gives the exact forward reuse distance for every execution of a memory instruction. Here, the additional advantage of being able to select cache hints per access, instead of per instruction,

is measured. In the experiments, we generated IA-64-like cache hints, using the second method described in section 5.3.1, based on the calculated FRD. A single cache level was simulated, which reacts similarly to cache hints as the Itanium and Itanium2 processors do. When the hint indicates temporal locality, the data is placed in the cache and marked as most recently used. When the hint indicates no temporal locality, the line is still brought into the cache, to exploit potential spatial locality. However, in order not to throw out too much data with temporal locality, the line is marked as the next to be replaced.

The relative miss rates for the programs compiled without hints, with static hints (per instruction) and with dynamic hints (per access) are shown in table 5.4. The table shows that on average static hints reduce the number of misses by 5.14% and dynamic hints reduce misses by 10.34%. The program which profits most from switching from static to dynamic hints is Cholesky. The reason is that it is the program with the most irregular reuse patterns. Since the same instruction requires different hints, static hints cannot improve the cache behavior. On the other hand, dynamic hints result in a cache miss reduction. For some programs (especially tomcatv), static hints improve cache behavior better than dynamic hints, which indicates that reuse distance based hint selection is not optimal. However, for the programs under consideration, dynamic hints never increase the number of misses, compared to the standard LRU replacement policy.

As described in section 5.3.2, generating code so that with each memory access, the exact forward reuse distance is associated, can be costly. In table 5.5, the overhead of both exact target hints and hints based on the forward reuse distance of the dominant domains (see section 5.3.2) is shown. Since dynamic hints require a small ISA-extension (see second method in section 5.3.1), this code cannot directly be executed on an Itanium processor. In order to measure code size and execution time, the programs with reuse distance calculation for every memory access were compiled without generating the actual hint, i.e. `ld r5=[r6]` is generated instead of `ld r5=[r6], frd=r7`. Table 5.5 shows that exact hints lead to a code size increase of 22 times (as measured in byte-size of the compiled object-file) and an execution time increase of 63.18 times (as measured by performance counters). However, when only taking the dominant reuse distance polynomials (see definition 29) into account, there's no execution time overhead on average, and the code size increases by only 17%. The average instructions executed per cycle increases by about 5%, which indicates that reuse

	LRU	LRU+static hints		LRU+dynamic hints	
program	miss rate	miss rate	reduction	miss rate	reduction
vpenta	31.56%	25.57%	*19.00%*	25.57%	*18.99%*
mxm	3.20%	3.17%	*0.87%*	3.20%	*0.00%*
liv18	68.46%	63.13%	*7.78%*	61.91%	*9.57%*
cholesky	19.81%	25.48%	*-28.59%*	17.94%	*9.43%*
jacobi	14.32%	14.32%	*0.02%*	14.32%	*0.00%*
gauss-jordan	11.90%	8.94%	*24.82%*	7.81%	*34.37%*
tomcatv	9.22%	8.11%	*12.05%*	9.22%	*0.00%*
average	22.64%	21.25%	***5.14%***	20.00%	***10.34%***

Table 5.4: The cache miss rates for a 4-way set associative 16KB cache with 32 bytes per line. The first column indicates the program name; the second column the miss rate with LRU replacement; the third and fourth column show absolute miss rate and miss rate reduction relative to LRU with the static cache hints; the fourth and fifth column shows miss rate and miss rate reduction the reduction with dynamic cache hints.

program	exact		dominant domains		
	code size	exec. time	code size	exec. time	rel. IPC
vpenta	4.81	2.48	1.11	1.02	0.98
mxm	1.83	34.45	1.01	1.00	1.00
liv18	47.10	55.36	1.15	1.02	1.16
cholesky	2.17	5.77	1.34	0.98	1.22
jacobi	2.65	10.69	1.01	0.99	0.89
gauss-jordan	2.71	72.62	1.15	1.01	1.01
tomcatv	92.72	260.88	1.39	1.00	1.08
average	22.00	63.18	1.17	1.00	1.05

Table 5.5: The overhead of dynamic computation of forward reuse distance and corresponding target cache hints, as compared to the original program without hints. The first two columns show the code size and execution time when generating exact target hints. The third and fourth column show the relative code size and execution time, when only taking into account the dominant domains (see section 5.3.2). The last column shows the relative IPC (instructions per cycle), of the dominant domains-version as compared to the original program without cache hints.

distance calculation is responsible for an extra 5% of dynamic executed instructions. However, these calculations are done mostly in parallel with other instructions, and are scheduled in unused instruction slots by the compiler, leading to no execution time overhead.

Table 5.6 shows that using only the dominant domains results in a very low number of wrongly predicted reuse distances. For only 0.6% of the accesses, the calculated reuse distance was wrong.

When the calculated reuse distance is wrong, it still leads to correct replacement decisions as long as it falls on the same side of the cache size as the actual reuse distance. Table 5.6 shows that only 0.05% of the accesses lead to an incorrect replacement decision.

Figure 5.11 plots relative code size and execution time overhead of dynamic target hints against the size of the original programs. The plot shows that execution time overhead remains lower than 3%, no matter the size of the program, resulting in good scalability. For code size, the plot shows that the largest program results in the largest code size overhead.

program	% wrong rd	% wrong hint
vpenta	0.31%	0.12%
mxm	0.50%	0.12%
liv18	0.21%	0.01%
cholesky	0.37%	<0.01%
jacobi	0.24%	0.03%
gauss-jordan	0.08%	<0.01%
tomcatv	2.52%	0.09%
average	0.60%	0.05%

Table 5.6: Percentage of wrong forward reuse distances and wrong cache hints, when using only the dominant polynomials.

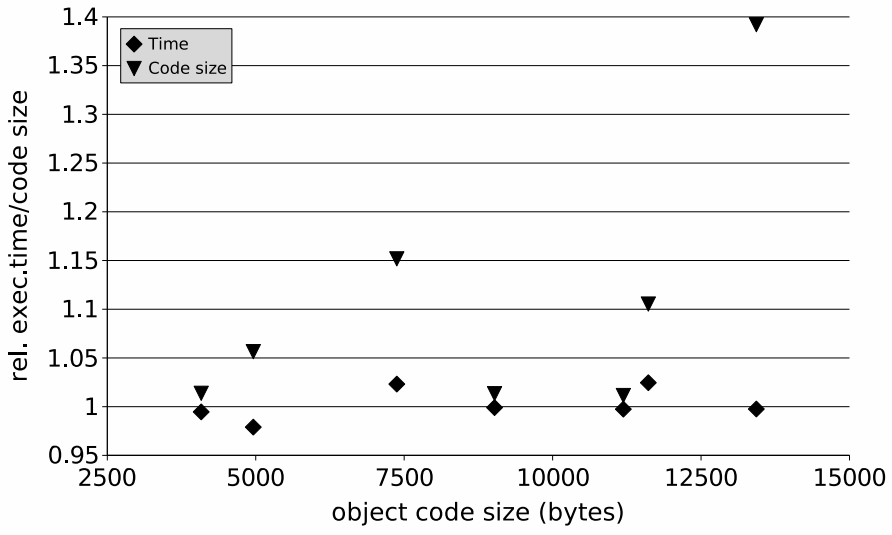


Figure 5.11: Scalability of code size and execution time overhead for dynamic target hints based on the dominant domains.

profile-based static hints	analysis-based dynamic hint
applicable for all programs	only applicable to programs representable in the polyhedral model
Optimized for a single data size	optimized for all possible data sizes
Only a single hint per instruction	The hint is computed for each individual memory access.
Tiny code overhead (<1%)	17% code size increase on average
Compatible with IA-64 ISA	Small extension to IA-64 ISA is necessary for generating efficient code

Table 5.7: Comparison of profile-based static hints versus analysis-based dynamic hints.

5.4.4 Discussion

The results of the experiments show that both static and dynamic hints can reduce the number of cache misses. In table 5.7, the main differences between the static hint and the dynamic hint selection are given. For programs in the polyhedral model, either static or dynamic hints can be generated. When an ISA is available which allows an efficient selection of dynamic cache hints (see section 5.3.1, second method), the analytical method is preferred, since (a) it optimizes for all possible input; (b) for every memory access the best hint is indicated; (c) the resulting binary is optimized for all possible cache sizes since the actual cache size isn't used in the compiler. On the other hand, when code size is of primary importance, e.g. in embedded systems, static hints might be preferred. Furthermore, in embedded systems the data size and the cache size is often fixed at compile time, so dynamic hints loose some of their advantages in that context.

Both static and dynamic hints are selected based on the reuse distance. One might wonder what the limitations are of reuse distance-based selection. A first limitation of the reuse distance is that it measures temporal locality, whereas caches also exploit spatial locality. The spatial locality is taken into account in the following ways:

- For the profile-based method, the term "memory location" in the

reuse distance definition is interpreted as being a "memory line". As such, the profile-based method measures reuse distance as the number of memory lines accessed between two accesses to the same memory line. In this way, spatial locality is taken into account, since two consecutive accesses to the same cache line have reuse distance 0.

- For the analytical method, the term "memory location" in the reuse distance definition is interpreted as being an array element. As such, spatial locality is not explicitly taken into account. Taking spatial locality into account in the formulas would result in a number of difficulties:

  1. The formulas become too complex for the current polyhedral tools to manage them (see section 4.6.2).
  2. If the size of the arrays is only known by parameters (e.g. `DIMENSION A(N,M)` in Fortran), the formulas would require quadratic expressions, resulting in formulas that don't fit in the polyhedral model.
  3. Even if the above two difficulties could be solved, then the exact solution of the reuse distance equations might become too complex to handle effectively.

  The analytical calculation is used to generate dynamic hints, which indicate at which cache levels temporal locality is lacking (e.g. `.nta` means "no temporal locality at all cache levels"). In order to capture potential spatial locality exhibited by these memory accesses, the complete memory line is always fetched in the cache even if the hint specifies no temporal locality. The difference with LRU is that the line is indicated as next to be replaced. The effectiveness of this method is indicated by the cache miss reductions shown in table 5.4, where this method is applied to numerical programs where the arrays are mostly traversed with stride 1, i.e. with very good spatial locality.

The basic assumption of reuse distance-based cache hint selection is that all data between use and reuse is fetched into the cache (see lemma 1), and is replaced with LRU policy. However, after hints have been introduced, intervening data with a non-temporal hint is replaced earlier. Even though this effect is not taken into account by the reuse distance, selecting cache hints based on the reuse distance assures that

cache hit rates are at least as good as under the LRU replacement policy [89]. A selection scheme which improves over reuse distance based selection would need to measure not only the amount of data accessed between use and reuse, but also what data is accessed, and what cache hints are assigned to it. For the profile-based method, this would result in having to record a huge amount of profile information. Furthermore, even if it is recorded which references and accesses occur between any two accesses to the same data, another problem arises: the selection of a cache hint for reference a might influence the cache hint selection for reference b, which in turn might influence the cache hint selection of reference a, resulting in a global optimization problem. In contrast to a method which would take these effects into account, the reuse distance based selection is simple and effective. Nonetheless, it would be interesting to look for practical methods which also take the interactions of cache hints into account.

5.5 Related Work

Source hints hide memory latency and target hints steer cache replacement decisions from the software. Most related work only focusses on one of these optimizations.

The best-known optimization to hide memory latency is prefetching [179], which can be performed either by the software or by the hardware. Most proposed software prefetching methods [42, 47, 116, 125, 126, 154, 196, 201] insert a prefetch instruction in the program, based on some form of locality analysis. In contrast, source hints do not insert extra instruction; they merely influence the generated instruction schedule. In comparison to prefetching, this results in less instructions. The advantage of prefetching is that it doesn't occupy registers during the long latency fetch from memory.

A few proposals similar to source hints have been made. In [81], Grun et al. propose to hide the latency of numerical programs by loop unrolling and shifting. In [133], Ozawa et al. classify load instructions as normal, list or stride accesses, based on a compile-time analysis. List and stride accesses are maximally hidden because they are suspected to cause most cache misses.

In recent years, a number of proposals have been made to steer the cache replacement policy by software [89, 152, 166, 174, 186, 197]. Tyson et al. [174] were one of the first to propose software-based cache

bypassing. When the cache hit-rate of a memory instruction is lower than a threshold, that data is not allocated in the cache. In [89], Jain et al. propose keep and kill instructions. The keep instruction locks data in the cache, while the kill instruction indicates it is the first candidate for replacement. In [186], Wang et al. propose to extend each cache line with an evict me-bit which is set by software, after a reuse vector-based locality analysis. In [197], Itanium cache hints are selected based on formulating the cache allocation problem as a knapsack problem. In [166], small extra L1-caches are proposed that can only be used by selected instructions, based on a region-analysis in the compiler. Similarly, in [152], separate temporal and spatial caches are proposed. Software instructions control wether data is allocated to the temporal or the spatial cache, after a locality analysis.

All these proposals are similar to static cache hints: they select a single caching strategy per instruction, even though the different executions of the instruction might exhibit different degrees of locality. In contrast, the dynamic cache hint selection communicates cache replacement decisions to the software which are optimized for each individual memory access.

5.6 Summary

Cache hints in EPIC architectures occur in two kinds: *source* and *target* hints. Source hints communicate the expected latency of memory instructions to the instruction scheduler, leading to a potentially improved overlap between computation and data fetch. Target hints communicate the forward locality of memory accesses to the processor, so that the replacement decisions in the cache can be improved. The hints are selected based on the locality of the memory accesses, as measured by the reuse distance. The source hints selection is based on the backward reuse distance, whereas the target hints are selected based on the forward reuse distance.

Next to the classification into source and target hints, an orthogonal classification of hints has been presented: *static* and *dynamic* hints. Static hints are hints which are fixed for all executions of an instruction. On the other hand, dynamic hints specify a hint tailored to each individual execution of a memory instruction.

The static hint selection scheme is based on profiled reuse distance distributions (as discussed in chapter 3). On the other hand, the dy-

namic target hint selection is based on analytically calculated reuse distances (see chapter 4). Exactly evaluating the reuse distance at each iteration gives rise to large overheads. However, when only taking the dominant reuse distance polynomials into consideration, the execution time overhead is eliminated, while the correct dynamic cache hints are generated for 99.95% of the memory accesses. Furthermore, the accuracy can be improved further with little overhead by peeling of the first or last loop iterations, where different reuse distance polynomials describe the forward reuse distance.

The experiments show that the source hints are most effective in the presence of a powerful instruction scheduling phase that can reschedule instructions over quite long distances, such as software pipelining. The static target hint selection results in up to 17% speedup, due to improved cache replacement decisions. The source hints result in speedups of up to 56% due to improved latency hiding. Dynamic target hints reduce the number of cache misses with up to 34%, with an average of 10%. Static target hints, on the other hand, result in a cache miss reduction of only 5%. This highlights the importance of generating dynamic hints, for which the hint is specific for each individual memory access.

Chapter 6

Cache Bottleneck Visualization

For many programs, current compiler techniques cannot adequately remove a significant portion of the cache misses. As seen in section 3.3, state-of-the art automatic optimizations are unable to reduce the number of long reuse distances.

Shortening long reuse distances requires an overview over millions of dynamically executed instructions. A powerful interprocedural analysis is needed to obtain this global program overview. Even if a compiler can pinpoint uses and reuses at long distances, it has to perform the daunting task of moving the use and reuse millions of dynamic instructions closer together, and still guaranteeing correct execution. With current program languages such as C, poisoned with dangling pointers, a compiler can only perform such optimizations in limited contexts, such as in singly nested loops (e.g. loop tiling), or a consecutive number of loops (e.g. loop fusion). Basically, to remove the longest reuse distances, a global overview of the complete program execution is needed.

In this chapter, it is presumed that a program's source code doesn't contain enough information for the compiler to correctly perform code transformations that reduce long reuse distances. Therefore, the task of shortening reuse distances is delegated to the programmer. However, a programmer also hasn't a clear view on the cache bottlenecks in his program. In this chapter, a visualization is proposed, which indicates the uses and reuses that are responsible for capacity misses. In contrast to earlier cache bottleneck visualizations, the proposed visualization not only shows the place where cache misses occur, but also hints the programmer at how the cache misses can be removed.

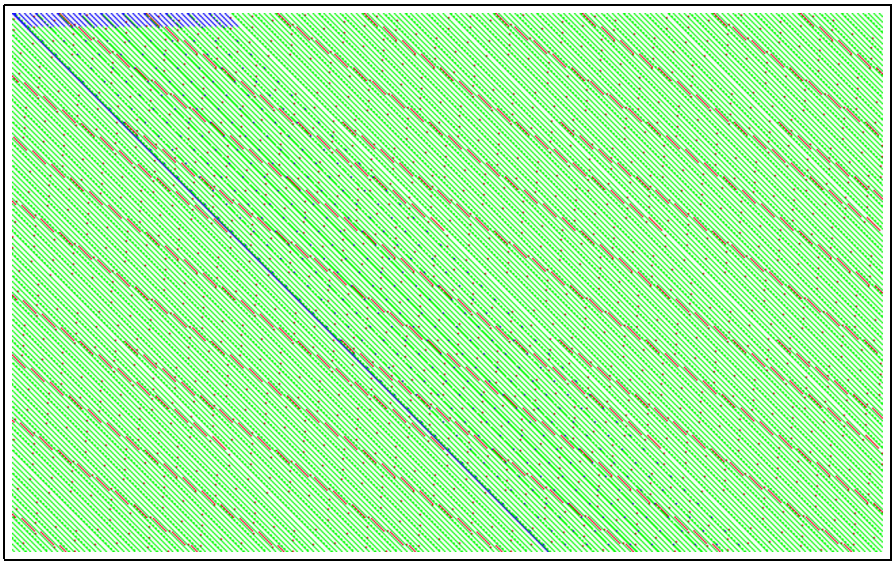
In the first section, a more traditional visualization is presented, which shows cache bottlenecks as a visual pattern. However, just like all earlier proposed visualizations, it doesn't clearly hint the programmer at how to resolve those misses. In the following sections, a new visualization based on pinpointing long reuse distances is presented, that directs the programmer into reducing long reuse distances. As a result of the shortened reuse distances, the cache behavior of the program is improved for a wide range of processors and cache hierarchies, i.e. the optimization is portable over a wide variety of machines.

6.1 Access Stream Visualization

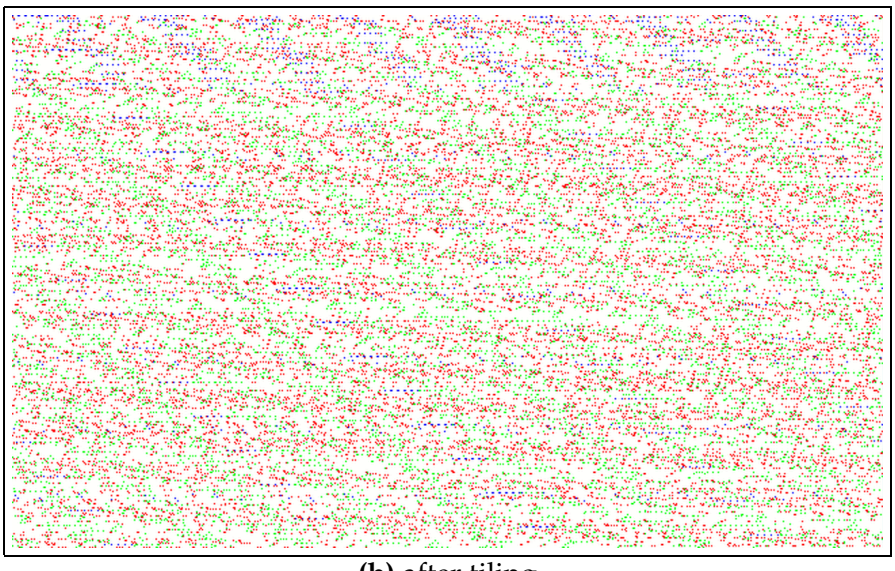
The visualization presented in this section has been developed in collaboration with Yijun Yu [198]. The presented visualization shows the cache behavior of individual memory accesses. It is used to show the effect of program optimizations on cache behavior, and can be used to guide the programmer to remove conflict misses. However, it will be indicated that the visualization is not very helpful in removing capacity misses (just like earlier visualizations [18, 31, 78, 107, 118, 124, 178] are not helpful in removing capacity misses, as discussed at the end of this section). In the next section, a visualization is presented which overcomes the shortcomings of the existing tools, and helps programmers to remove capacity misses.

The access stream visualization plots a pixel for each memory access. The pixel can have four colors: white for a cache hit, blue for a cold miss, green for a capacity miss, red for a conflict miss. The pixels corresponding to consecutive memory accesses are plotted next to each other horizontally. When the end of the screen is reached, the plotting is continued at the beginning of the next line. In this way, the 2-dimensional screen area can be fully used, with colors encoding the type of misses. Furthermore, the pixels on the same line encode memory access in a short time window, while the vertical direction encodes cache behavior over a larger time frame. Examples for three programs are shown. The cache behavior of a matrix multiplication is shown in figure 6.1(a), a fast Fourier transform is visualized in figure 6.2(a), and the tomcatv program is plotted in figure 6.4(a). In each plot, a direct mapped 1KB cache is assumed.

The plots show that for the matrix multiplication, capacity misses dominate; in the FFT code, both conflict and capacity misses are significant; while in tomcatv, conflict misses dominate. Cold misses only

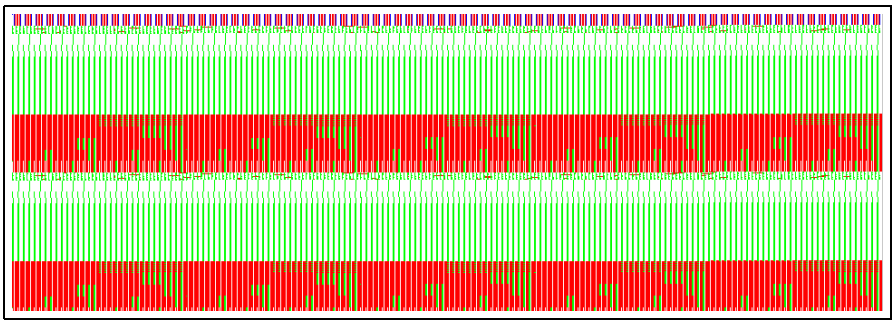


(a) original

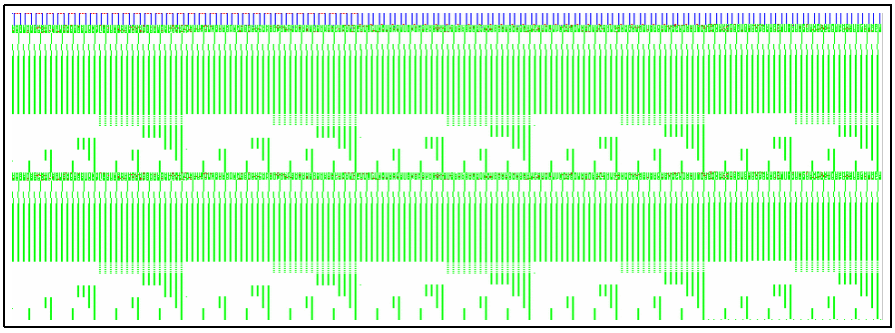


(b) after tiling

Figure 6.1: Matrix multiplication trace visualization



(a) original

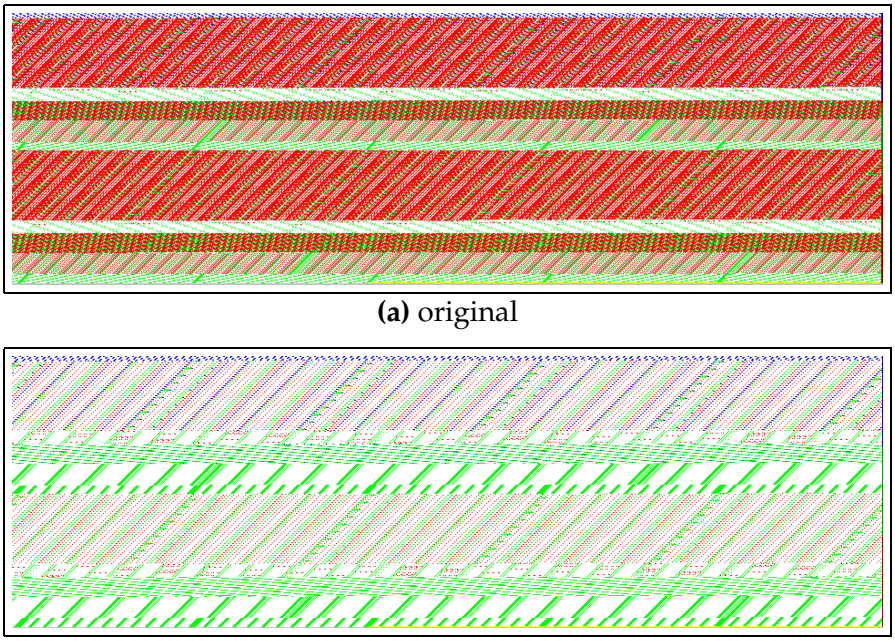


(b) after padding

Figure 6.2: FFT trace visualization

```
      DO     60    J = 2,N-1
        DO     50    I = 2,N-1
          XX = X(I+1,J)-X(I-1,J)
          YX = Y(I+1,J)-Y(I-1,J)
          XY = X(I,J+1)-X(I,J-1)
          YY = Y(I,J+1)-Y(I,J-1)
          A  = 0.25D0  * (XY*XY+YY*YY)
          B  = 0.25D0  * (XX*XX+YX*YX)
          C  = 0.125D0 * (XX*XY+YX*YY)
          AA(I,J) = -B
          DD(I,J) = B+B+A*REL
          PXX = X(I+1,J)-2.D0*X(I,J)+X(I-1,J)
          QXX = Y(I+1,J)-2.D0*Y(I,J)+Y(I-1,J)
          PYY = X(I,J+1)-2.D0*X(I,J)+X(I,J-1)
          QYY = Y(I,J+1)-2.D0*Y(I,J)+Y(I,J-1)
          PXY = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
          QXY = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
          RX(I,J) = A*PXX+B*PYY-C*PXY
          RY(I,J) = A*QXX+B*QYY-C*QXY
50      CONTINUE
60    CONTINUE
```

Figure 6.3: Highlighted Tomcatv source code



(a) original

(b) after padding

Figure 6.4: Tomcatv trace visualization

occur at the beginning of the program execution. After tiling, many of the capacity misses are eliminated for the matrix multiplication (see figure 6.1(b)).

In the FFT-code, the conflict misses are caused by accesses to the main matrix with stride 2^x, which are inherent in the FFT algorithm. These are resolved by padding the array, so that the strides become $2^x + c$, with c being a small value larger than the cache line size.

For tomcatv, the repeating vertical pattern in figure 6.4(a) reveals the structure of the source code: an outer loop that executes two iterations. The outer loop contains 5 inner loops. The first inner loop, which is displayed in figure 6.3, generates most memory accesses and also generates most conflict misses. In figure 6.3, the memory references in the first inner loop of the tomcatv programs are colored, according to the kind of cache misses they exhibit. Cold misses are indicated by blue, conflict misses by red and capacity misses by green. The different executions of a reference can generate different kinds of misses, and therefore the color of a reference is the weighted average of the colors corresponding to the misses it generates.

The conflicts between arrays X and Y were removed by enlarging the arrays from size 513×513 to size 513×520, so that elements in the same row no longer map to the same cache line. This optimization leads to a 50% speedup on a pentium III 550Mhz machine.

The access stream visualization described above allows to visually see the cache misses. However, the underlying causes of the misses are not pinpointed by the visualization. Conflict misses can be removed by finding the array pairs which generate conflicts, and applying intra- or inter-array padding.

In order to resolve capacity misses, however, the use and the reuse of data should be brought closer together, and this long-distance use and reuse is not clear from the visualization. All earlier proposed visualizations are limited in one way or another at pinpointing causes of capacity misses. The visualizations that have been proposed in the literature either show the contents of the cache itself (i.e. *cache-centric*), or the program objects (such as source lines, and variables) that cause the cache misses:

cache-centric The Cache Visualization Tool CVT [178] graphically shows cache lines as colored rectangles, where the color of the cache line indicates which array variable is currently held at that line. A memory trace is played back and the programmer sees how the contents of the cache changes dynamically. In Rivet [31] the cache contents is also shown, and changes dynamically as the memory access trace is processed. These visualizations give an idea of the short-term dynamics of the cache contents. In [178], it is shown that the short-term cache dynamics can help the programmer in pin-pointing causes of conflict misses. However, it is not very useful in helping the programmer to reduce capacity misses, since the cause of capacity misses (use and reuse) are far apart in time, and the visualization doesn't help the programmer in recognizing long distance reuse patterns.

program-centric In contrast to the cache-centric visualization, different authors have proposed program-centric visualizations. SIP [18], Mtool [78], HPCView [124], Cprof [107], Memspy [118] and VTune [54] all indicate the source code lines at which cache misses occur, or the variables that cause cache misses. As a result, the programmer knows what data accesses result in misses (conflict or capacity), but he is not supported in finding a program trans-

formation to reduce capacity misses. The programmer is supported in reducing conflict misses through the identification of the arrays that incur conflict misses, which hints at transforming the data layout of those arrays.

All of the above visualizations are limited in helping the programmer in resolving capacity misses. At best, they show the source lines where capacity misses occur, or the variables that lead to capacity misses. In the next section, a visualization is proposed which overcomes these limitations.

6.2 Low-Locality Reuse Visualization

6.2.1 Platform-Independent Cache Optimizations

When the task of optimizing the cache behavior is delegated to the programmer (and hence requires costly human time and effort), it should be tried to make the cache optimizations portable. In other words, when the program runs on a different processor with a different memory hierarchy, the cache optimizations should still be effective. Furthermore, useful programs tend to be used for a long time, and be recompiled on new computers, years after they have been written. Therefore, a good cache bottleneck visualization should guide the programmer into making optimizations which reduce cache misses on a wide variety of architectures. Earlier visualizations all aim at optimizing the program for a single specific cache architecture (since they only show cache misses in one specific cache configuration). If the optimizations based on those visualizations result in cache improvements on different architectures, it is merely a lucky coincidence.

Since capacity misses typically dominate, the focus is on eliminating capacity misses. Capacity misses or their slowdown effect can be reduced in four different ways.

1. The first way is to *eliminate the memory accesses with poor locality* altogether. Sometimes one can choose a different data structure, or a different algorithm which enables this. An example of such an optimization is the following. Consider a 2-dimensional array in C, which is allocated as an array of arrays: `double** A; A[i][j]=0;`. The code `A[i][j]` results in two load instructions, one to fetch `A[i]`, the second to fetch `A[i][j]`. If the load

instruction `A[i]` causes capacity misses, it can be eliminated by transforming the array into a single-dimensional array: `double* A; A[i*N+j]=0;`.

2. A second way is to reduce the distance between use and reuse for long reuse distances, so that it becomes smaller than the cache size. This can be done by reordering computations (and memory accesses), so that *more temporal locality* is achieved. Examples of such optimizations are loop fusion and loop tiling. The general idea behind this family of optimizations is to try to do as much useful computations as possible on the data while it is in the cache.

3. A third way is to *increase the spatial locality*. This is most easily done by rearranging the data layout. Examples of such optimizations are array transposition and field reordering. The general idea for increasing spatial locality, is to rearrange the data layout so that variables which are accessed close together end up in the same memory line. Consider two consecutive accesses to two different variables. If the variables lay in different memory lines, the backward reuse distance of the second access might be very large. However, if both variables are located on the same memory line, the reuse distance of the second access is 0.

4. If neither of the three previous methods are applicable, it might still be possible to improve the program execution speed, not by eliminating the capacity misses, but by *hiding their latency with independent parallel computations*. The best-known technique in this class is prefetching.

6.2.2 Locality Metrics

A large reuse distance indicates that the reuses have low temporal locality, and therefore a cache is unlikely to retain the data between the reuses. However, the data is reused, and therefore, the reuse could in principle be exploited by keeping the data in a fast memory. A large reuse distance indicates inefficient temporal exploitation of reuses.

Next to exploiting temporal locality, caches also exploit spatial locality. The following metric is used to measure the amount of spatial locality:

Definition 30. *The* **memory line utilization** *of an access a to memory line l is the fraction of l which is used before l is evicted from the cache.* □

When the backward reuse distance of access a is larger than the cache size, it results in a cache miss. Consequently, a fetches memory line l into the cache. If the memory line utilization of a is less than 100%, not all the bytes in l are used, during that stay in the cache. Therefore, at access a, some useless bytes in l were fetched. As a result, the potential benefit of fetching a complete memory line was not fully exploited. The memory line utilization metric indicates how much the spatial locality could be improved.

6.2.3 Implementation

Instrumentation and Locality Measurement

In order to measure the reuse distance and the memory line utilization, the program is instrumented to obtain the memory access trace. The ORC-compiler [132] was extended, so that for every memory access instruction, a function call is inserted. The accessed memory address, the size of the accessed data and the instruction generating the memory access are given to the function as parameters. The instrumentation ensures that the function is called for every memory access.

The instrumented program is linked with a library that implements the function which is called on every memory access. The reuse distance and the memory line utilization is calculated on-line for each memory access, so that the memory trace doesn't have to be stored. For every pair of instructions (r_1, r_2), the distribution $\mathsf{RDD}(r_1, r_2)$ is recorded (see definition 9, on page 92). In order to reduce the amount of information which needs to be recorded, the accesses are categorized into sets with power of 2. For example, if the backward reuse distance is 18, it is recorded to be in the set of reuse distances between 2^4 and 2^5.

The memory line utilization of a reuse pair is measured as follows. A fixed cache size $\mathsf{CS}_{\mathsf{min}}$ is chosen to be the minimal cache size of interest. Whenever an access a, accessing memory line l, has a backward reuse distance longer than $\mathsf{CS}_{\mathsf{min}}$, the memory line utilization of that access is measured. On every access to the line l, it is recorded which bytes were used. When line l is evicted from the fully-associative cache of size $\mathsf{CS}_{\mathsf{min}}$, the fraction of the bytes which were accessed during that stay of line l in cache of size $\mathsf{CS}_{\mathsf{min}}$ is recorded. In the experiments in

section 5.4, $CS_{min} = 2^8$ cache lines of 64 bytes each = 16 KB.

At the end of the program execution, the recorded reuse distance distributions, together with the average memory line utilizations are written to disk. In our implementation, the data is stored in an XML-format, which allows easy manipulation with XSLT-scripts.

A large overhead can result from instrumenting every memory access. To reduce the overheads, sampling was used so that reuses are measured in bursts of 20 million consecutive accesses, while the next 180 million accesses are skipped. In that way, for only 10% of the memory accesses, the reuses are actually calculated. Furthermore, the memory overhead is further reduced by measuring the reuse distances with a memory line granularity of 64 bytes. In the experiments, a slowdown between 15 and 25 was measured when compared to uninstrumented execution. The instrumented execution consumes about twice as much memory as the original, due to bookkeeping needed for reuse distance calculation. The overheads are quite reasonable, considering the detailed level of information that is recorded, and that using this information, an average speedup of 3.06 was obtained.

Visualization

Theorem 1 on page 93 indicates that only the reuses whose distance is larger than the cache size generate capacity misses. Therefore, only the reuse pairs with a long reuse distance (=low locality) are shown to the programmer. Furthermore, in order to guide the programmer to the most important low-locality reuses, only the instructions pairs which generate at least 1% of the long reuse distances are visualized. The visualization shows the reuse pairs as arrows drawn on top of the source code. A label next to the arrow shows how many percent of the long reuse distances were generated by that reuse pair. Furthermore, the label also indicates the memory line utilization of the cache missing accesses generated by that reuse pair. A simple example is shown in figure 6.6.

In the prototype implementation, an XSLT-script translates the XML-file containing the measured reuse distances and memory line utilizations into input to the VCG [153]-tool. Then, VCG is used to display the graph. The examples of the visualization are drawn by hand, in order to have a denser visualization than the one generated by the VCG-tool (see figures 6.6, 6.7 and 6.8). An example of the visualization

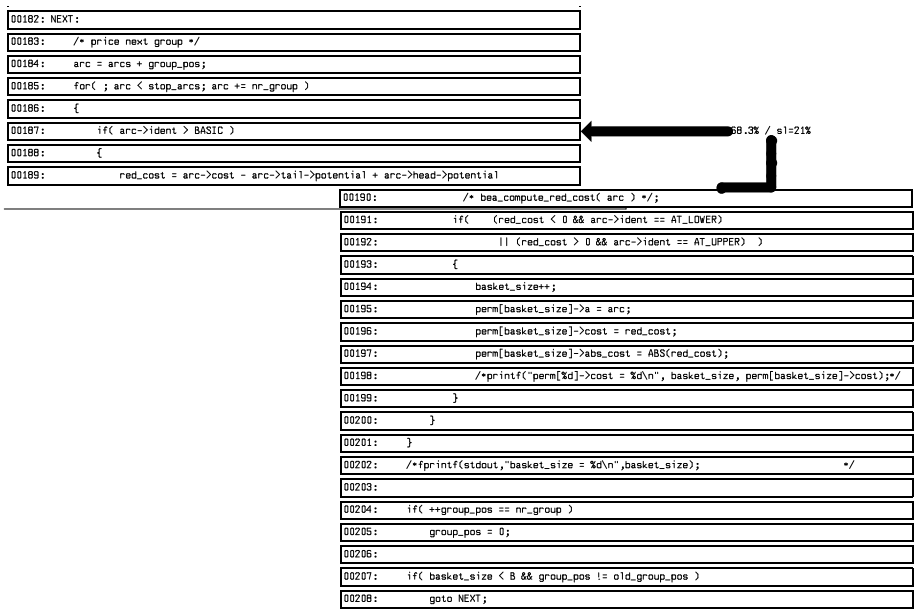


Figure 6.5: VCG-generated visualization of major long distance reuses in mcf. There is a single pair of references which produce most of the long reuse pairs (68.3% of all the long reuse pairs are generated by these references, as indicated by the edge label). The second number (sl=21%) indicates that the cache-missing reference (on line 187) has a memory line utilization of 21%.

generated by the VCG-tool for the mcf-program is shown in figure 6.5.

6.3 Experiments

In order to evaluate the benefits of visualizing low-locality reuses, the three programs from SPEC2000 with the highest cache bottlenecks were considered: 181.mcf, 179.art and 183.equake (see figure 1.1 on page 63). Below, for each of the programs, the visualization of the main cache bottlenecks is shown. Also, it is discussed how the programs were optimized and what speed-up was measured on CISC, RISC and EPIC processor systems.

6.3.1 Mcf

The mcf-program solves single-depot vehicle scheduling problems occurring in the planning process of public transportation companies.

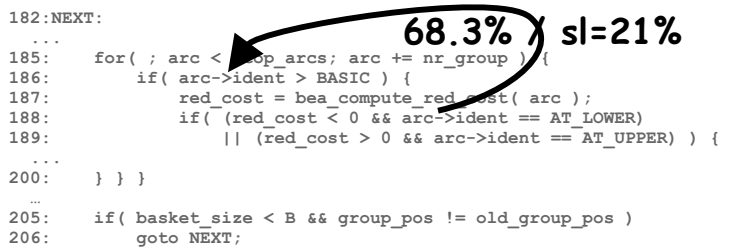


Figure 6.6: A zoom in on the major long reuse distance in 181.mcf. There is a single pair of instructions which produce most of the long reuse pairs (68.3% of all the long reuse pairs are generated by these references, as indicated by the edge label). The second number (sl=21%) indicates that the cache-missing instruction (on line 186) has a memory line utilization of 21%.

This problem is solved using a network simplex algorithm [165]. The main long reuse distances for the 181.mcf program are shown in figure 6.6. The figure shows that about 68% of the capacity misses are generated by a single load instruction on line 187. The best way to solve those capacity misses would be to shorten the distance between the use and the reuse. However, after analyzing the code a bit further, it shows that the reuses of `arc`-objects occurs between different invocations of the displayed function. So, bringing use and reuse together needs a thorough understanding of the complete program, which we do not have, since we didn't write the program ourselves. A second way would be to increase the memory line utilization from 21% to a higher percentage. To optimize the spatial locality, the order of the fields of the `arc`-objects could be rearranged.

However, this change leads to poorer spatial locality in other parts of the program, and overall, this restructuring does not lead to speedup. Therefore, we tried the fourth way to improve cache performance: inserting prefetch instructions to hide the miss penalty.

6.3.2 Art

The 179.art program performs image recognition by using a neural network. A zoom-out view of the main long reuse distances in this program is shown in figure 6.7. The function where these long reuses occur is `match`, in which the neural network is evaluated for a specific input image. Each node in the neural network is represented by a struct containing 9 `double`-fields, so the 9 fields representing a neuron are layed

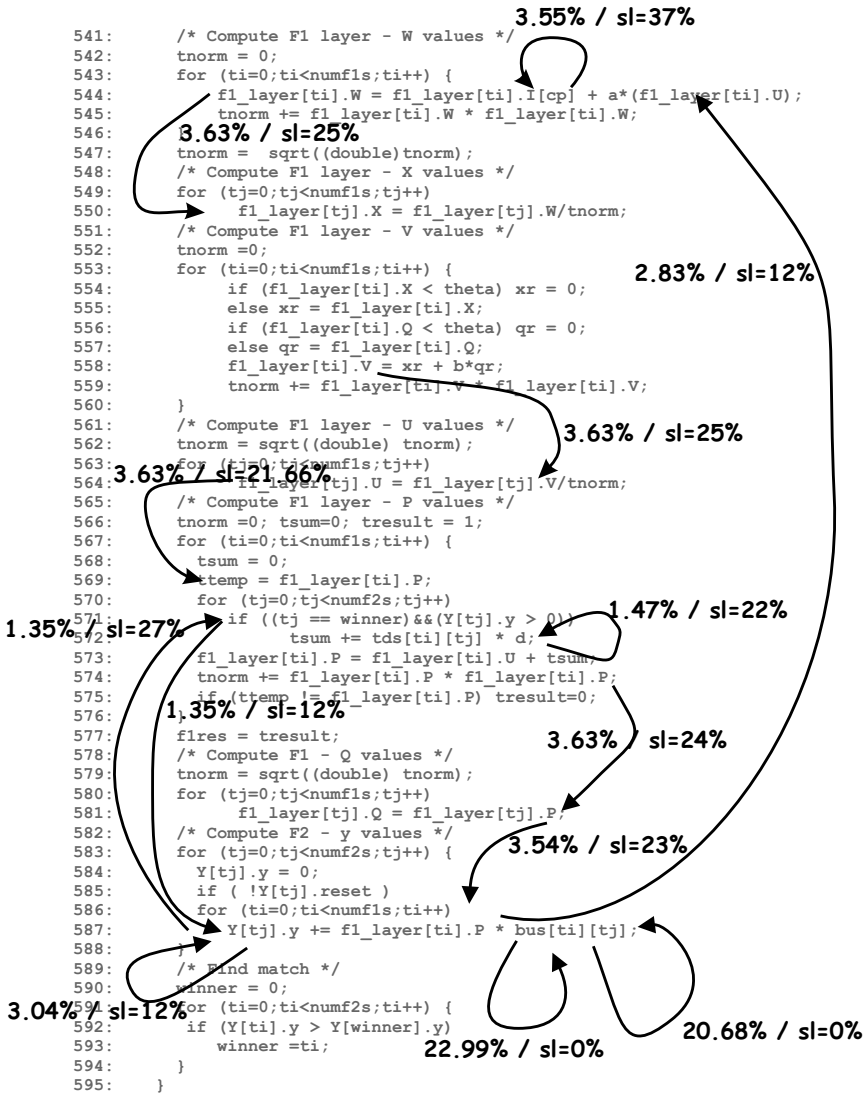


Figure 6.7: A zoom-out view of the major long reuse distances in 179.art

out in consecutive locations.

The visualization shows that the memory line utilization of most accesses with long reuse distance is low ($< 20\%$). The code consists of 8 loops, each iterating over all neurons, but only accessing a small part of the 9 fields in each neuron.

A simple data layout optimization resolves the spatial locality problems. Instead of storing complete neurons in a large array, i.e. an array of structures, the same field for all the neurons are stored consecutively in arrays, i.e. a structure of arrays. Besides this data layout optimization, also some of the 8 loops were fused, when the data dependencies allowed it and reuse distances were shortened by it. The obtained speedups after these optimizations are shown in figure 6.9.

6.3.3 Equake

The equake program simulates the propagation of elastic waves during an earthquake. A zoom-out view on the major long reuse distances in this program is shown in figure 6.8. All the long reuse distance pairs occur in the main simulation loop of the program. This main simulation loop has the following structure:

- Loop for every time step (line 447–512):
  - Loop to perform a sparse matrix-vector multiplication. (line 455–491)
  - A number of loops to rescale a number of vectors. (line 493–507)

Most of the long reuse distances occur in the sparse matrix-vector multiplication, for the accesses to the sparse matrix. The sparse matrix is a 3-dimensional array `K`. The matrix is symmetric, and only the upper triangle is stored. The array is declared as `double *** K;`. An access to an element has the form `K[Anext][i][j]`, leading to three loads. The number of memory accesses is reduced by redefining this array as a single dimensional array. The access above is transformed into `K[Anext*N*9+i*3+j]`, leading to a single load instruction.

Furthermore, after analyzing the code a little further, it is obvious that for most of the long reuse pairs, the use is in a given iteration of the time step loop and the reuse is in the next iteration. Therefore, in order to bring the use and the reuse closer together, some kind of tiling

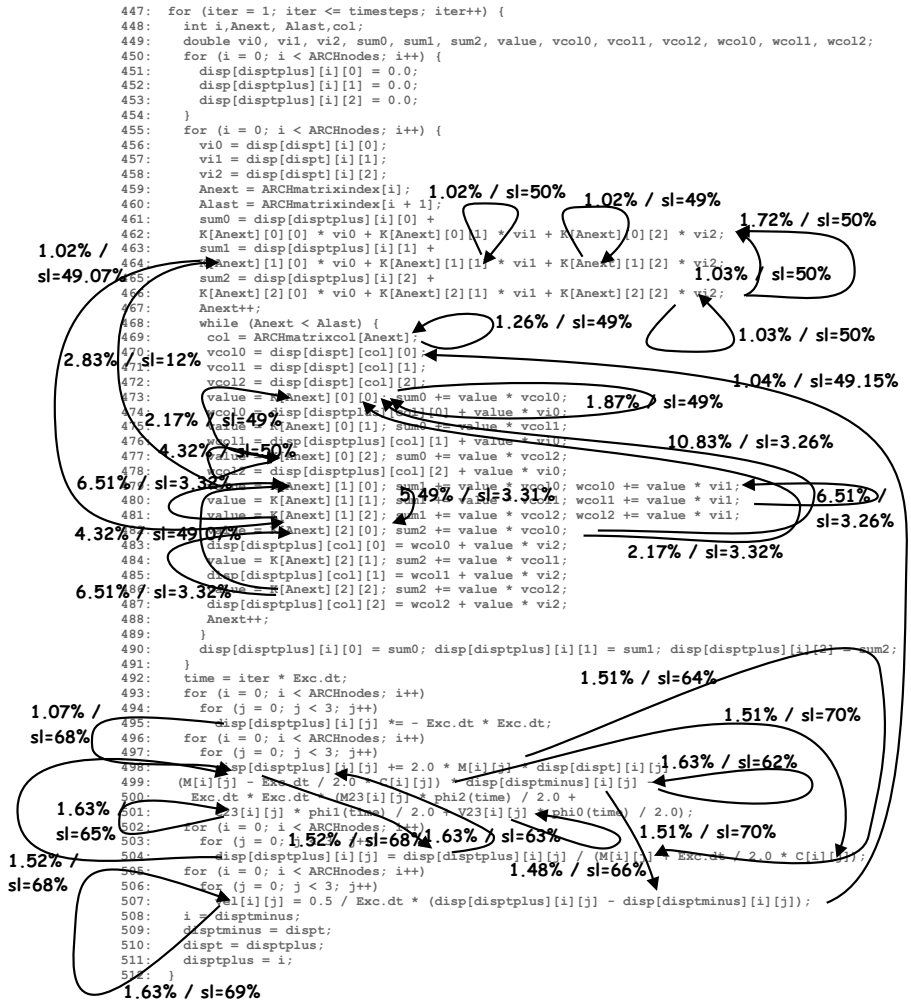


Figure 6.8: A zoom-out view of the major long reuse distances in 183.equake.

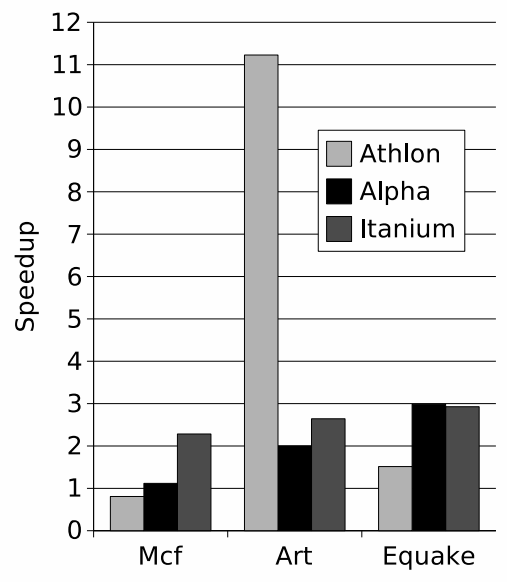


Figure 6.9: Speedups on different architectures.

transformation should be performed on the time-step loop (i.e. try to do computations for a number of consecutive time-steps on the set of array elements that are currently in the cache). However, the loop dependencies must be taken into account. To break dependencies, the sparse matrix was changed, so that all the elements were stored in memory, and not only the upper triangle. This allows to simplify the sparse matrix-vector multiply code, and remove some loop dependences. After this, it was possible to fuse the matrix-vector multiply loop with the loops which rescale the vectors, resulting in a single perfectly-nested loop. In order to tile this perfectly-nested loop, the structure of the sparse matrix needs to be taken into account, to figure out the real dependencies, which are only known at run-time. The technique described in [63] was used here to perform a run-time calculation of a legal tiling. The speedups obtained by this optimization are shown in figure 6.9.

6.3.4 Discussion

The original and optimized programs were compiled and executed on different platforms, in order to measure the performance portability of the optimizations: an Athlon PC, an Alpha workstation and an Itanium

processor	L1 (size,assoc)	L2 (size,assoc)	L3 (size,assoc)
Athlon XP 1800+	(64KB, 2)	(256KB, 16)	not present
Itanium 733Mhz	(16KB, 4)	(96KB, 6)	(2MB, 4)
Alpha 21264	(64KB,2)	(8MB, 1)	not present

Table 6.1: Cache sizes and associativity for the different processors.

server. The cache configurations of the different processors is shown in table 6.1. For the Athlon and the Itanium, the Intel compiler was used. For the Alpha 21264, Digital's Alpha compiler was used. All the programs were compiled with the highest level of feedback-driven optimization.

In figure 6.9, the speedups on the different processor systems are presented. In this table, it shows that most programs have a good speedup on most processors. The only exception is Mcf on the Athlon. Figure 6.10 show that the long reuse distances have been effectively diminished in both art and equake. In mcf (not displayed), the reuse distances were not diminished, since only prefetching was applied. Only on the Athlon, a slow-down is measured, probably because the hardware-prefetcher in the Athlon interferes with the inserted software prefetch instructions.

6.4 Related Work

The cache visualization can be categorized into cache-centric and program-centric visualizations. The cache-centric visualizations, such as CVT [178] and Rivet [31], show the contents of the cache at a given point in time. By showing the changes in the cache contents over time, effects such as cache conflicts become visible.

In contrast to cache-centric visualizations, program-centric visualizations, such as SIP [18], Mtool [78], HPCView [124], Cprof [107], Memspy [118] and VTune [54], show cache miss statistics of program objects such as source code lines and program variables. Many of these visualizations also aim at reducing the cache misses in multiprocessor systems. The low-locality visualization might also be extended to indicate multiprocessor cache behavior, by also considering consecutive reuses where the use and reuse occur on different processors. However, it is not clear for how far these reuses are platform-independent.

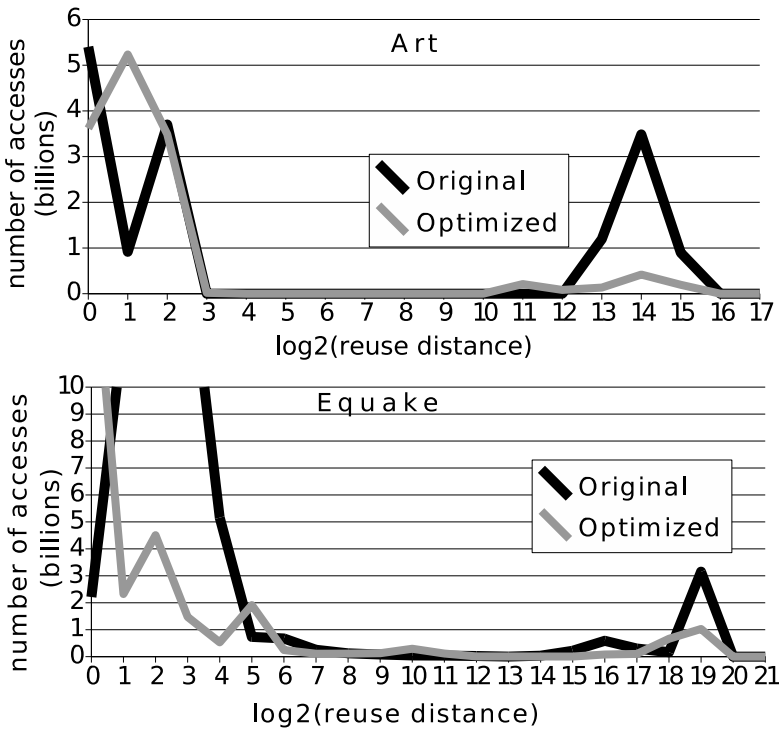


Figure 6.10: Reuse distance distributions before and after optimization.

The main limitations of the previous visualizations in comparison to the low-locality visualization presented in section 6.2 are:

- The cache behavior of only one specific cache or cache hierarchy is visualized. In contrast, the low-locality visualization indicates cache bottlenecks for a wide range of caches, by abstracting cache behavior using the reuse distance metric. Therefore, the low-locality visualization is the only one that explicitly steers the programmer to platform-independent cache optimizations.
- The previous optimizations merely indicate cache misses, but they don't hint the programmer in how to optimize the program. In contrast, the low-locality visualization indicates the previous use of a cache missing reuse. This gives the programmer a strong hint on how he can increase the locality of the cache miss.

Pingali et al. [136] have recently proposed a "computation regrouping" methodology which consists of a number of guidelines for cache optimizations at the source code level. They show speedups between 1.26 and 3.03 on a set of benchmarks optimized by hand. They also conclude that it is not clear how a compiler might perform similar optimizations, which indicates that compiler-driven cache optimization might never reach the same performance level as programmer-driven optimizations. This stands in contrast to low-level assembler optimizations, where compilers reach about the same optimization levels as programmers.

6.5 Summary

Removing capacity misses requires knowledge of the use and subsequent reuse causing the miss. However, these reuses are far apart, and a large overview of the program execution is needed. As discussed in chapter 3, in most cases, the analysis phases in the compiler are not powerful enough to obtain this overview, or the optimization phases cannot reorganize the program enough to reorder cache missing instructions over millions of dynamic instructions. Therefore, the cache optimizations need to be delegated to the programmer, who has a better understanding of the interactions in his program, and has more freedom to change algorithms than a compiler. However, the cache behavior of the program is also not clear to the programmer and he needs help to pin-point the causes of cache misses in his code.

In this chapter a visualization of long reuse distance pairs has been proposed, which indicates where and how temporal and spatial locality can be improved to reduce the number of cache misses. In contrast to earlier visualization methods, the long reuse distance visualization steers the programmer into performing platform-independent cache optimizations.

The visualization has been applied to the three SPEC2000 programs with the largest cache bottlenecks. Their cache behavior has been optimized by small source code changes, based on the visualized long reuse distances. The optimizations lead to a speedup of 3.06 on average, on a set of different platforms with varying cache hierarchies.

Chapter 7

Conclusion

In this chapter, the conclusions from this dissertation are summarized, and a structured overview of the main contributions is given. Furthermore, interesting topics for future research based on the results in this dissertation are highlighted.

7.1 Summary and Contributions

The processor-memory speed gap is growing at an exponential rate. To keep processors from being data-starved, ever better methods to exploit memory hierarchies need to be devised. For many programs and cache configurations, it shows that capacity misses dominate over conflict misses. Basically, the processor-memory gap can be bridged by methods at the hardware and microarchitectural level, at the compiler level or at the algorithmic level. At the microarchitectural level, only conflict misses can be addressed. In this dissertation, the focus is on compiler and algorithmic level optimization of locality and cache behavior, to reduce the dominating capacity misses. The main contributions of this research are in the following areas:

Cache Remapping The cache remapping method, as presented in chapter 2 and [20, 21], aims at completely removing processor stall due to cache misses. This is done by combining ideas from four categories of software optimizations: tiling reduces the number of capacity misses, conflict misses are removed by dynamically remapping data in the cache, cache hints are used to control cache replacement, and preloading in multiple threads

is performed to hide the latency of left-over main memory accesses. The program is conceptually divided in two threads: a computation thread that performs the original program computations, and a data fetch thread that is responsible for moving data between the cache and main memory. Both threads can be interwoven at compile time, so that cache remapping is also applicable for single-threaded processors.

Reuse Distance as a metric for Locality Before capacity misses can be eliminated, a good model of their causes is needed. In chapter 3 and in [28], the reuse distance metric is discussed. The reuse distance metric exactly predicts cache behavior for fully-associative caches. Own measurements and observations from the literature show that the reuse distance is also a good predictor of cache behavior for less-associative caches. Furthermore, the experiments indicate that state-of-the-art compiler technology is unable to reduce the number of long distance reuses. On the Spec95FP benchmarks, only 1% of the capacity misses were removed, while 30% of the conflict misses were eliminated. Combining this with the fact that at the microarchitectural level, capacity misses can only be removed by enlarging the cache, it seems that capacity misses are inherently more difficult to remove than conflict misses.

Reuse Distance Equations Next to profiling reuse distances, they can also be calculated analytically for programs that fit the polyhedral model, as presented in chapter 4 and [22, 158, 184]. The advantage over profiling is that the reuse distance for all memory accesses is described by a few Ehrhart polynomials. Furthermore, the equations allow to calculate reuse distances for all possible data input sizes, whereas profiling only measures reuse distances for one specific program execution.

The reuse distance calculation is based on the polyhedral program model. One of the corner stones in solving the equations is the computation of Ehrhart polynomials that describe the number of integer points in parametric polytopes. Three limitations of the state-of-the art interpolation method for counting Ehrhart polynomials are described. A new method based on Barvinok's decomposition method has been proposed, which solves the three limitations in the interpolation method.

Furthermore, the reuse distance equations have been extended to cache equations. Cache equations have been proposed several

times, but they could only be applied to very simple loops due to limitations in solving Presburger formula and counting integer points in parametric polytopes. In chapter 4, the problem with counting integer points in parametric polytopes has been solved by the method using on Barvinok's decomposition. The problem with simplifying Presburger formula resulting from cache equations has been identified, but no clear solutions are envisioned. It might be that general cache behavior is too irregular to be exactly modelled by a set of a few Ehrhart polynomials.

Cache Hint Generation The generation of source and target cache hints has been discussed in chapter 5 and [22, 23, 24, 158]. Static hints are selected based on profiled reuse distance distributions. They lead to a speedup of 10% on average on an Itanium processor. In contrast, dynamic hints are generated based on reuse distance equations. By only encoding the dominant Ehrhart polynomials, no execution time overhead results from evaluating the polynomials at run-time. Static target hints lead to 5% less cache misses, while dynamic target hints reduce the number of misses by 10%. The reuse distance equations are the first cache analysis method which allows dynamic cache hints with no execution time overhead, thanks to a compact representation of the locality of individual memory accesses by Ehrhart polynomials.

Cache Visualization Reducing capacity misses requires bringing reuses at long distance closer together and a large overview of program execution is needed. However, this is often hard for a compiler, and therefore, the task is delegated to the programmer. In chapter 6 and in [25, 27, 198], visualizations are proposed that show the programmer where low-locality reuses occur in his program. The visualization steers the programmer into optimizing the locality in a platform-independent way. As a proof of concept, the three programs from SPEC2000 with the largest cache bottlenecks have been optimized at the source code level. The optimized codes run 3 times faster on average on a variety of computer platforms with different processors and cache hierarchies.

7.2 Future Research Directions

A number of interesting directions for future research based on the work presented in this dissertation are summarized below:

- In the cache remapping method, the software controls which data ends up in the cache, and at which times data needs to be transferred between cache and main memory. In some new (mostly embedded) architectures, scratch pad memories are introduced. These scratch pads are fast and small on-chip memories. In contrast to caches, they must be explicitly managed by software, which can access them through a reserved range in the address space. It would be interesting to see how cache remapping could be employed in such an environment, so that the compiler automatically generates code that keeps the current working set in the scratch pad.

- Solving the reuse distance equations, as described in chapter 4, has been enabled by the new method to compute Ehrhart polynomials, based on Barvinok's decomposition. When the reuse distance equations are adapted to model cache behavior (i.e. longer cache line size and set-associativity), the corresponding Presburger formulas become more complex. Current tools and methods for converting these Presburger formulas into a set of polytopes are not powerful enough to handle them. Therefore, to get a practical method that computes cache behavior exactly for arbitrary caches, one of the following obstacles need to be solved:

  1. A method to translate parametric Presburger formula into a set of disjoint parametric polytopes.
  2. A method that is able to count parametric Presburger formula.
  3. A formulation of cache behavior which leads to simpler Presburger formula, so that current tools can solve them. Alternatively, a formulation might be devised in which the cache behavior is entirely described by parametric polytopes, without needing Presburger formula.

  However, cache behavior seems to be quite irregular, and little differences in parameters, such as line size and data layout can change the cache behavior substantially. Therefore, it might be

impossible to describe general cache behavior by a small set of Ehrhart polynomials.

Another way to reduce the complexity of the reuse distance calculation would be to somehow only compute the dominant reuse distance polynomial. However, it is not trivial to see how this could be done without also computing the non-dominant polynomials.

- The cache hint selection is based on the reuse distance. As such, it doesn't take into account the cache hint interactions. For example, when a cache hint for one instruction is chosen to be `.nta`, then it might become better to keep the data accessed by another instruction in the cache, even though its reuse distance is larger than the cache size. It is not clear how this interaction can easily be modelled for general programs. Even for polyhedral programs it is not clear how to efficiently select cache hints, when taking these interactions into account. It becomes even harder when dynamic hints are considered; even more so if the dynamic hint selection should have low code size and execution time overheads.

- The visualization of long distance reuses discussed in chapter 6 enables portable source code optimizations, leading to an average speedup of 3. However, the visualization itself can be improved further, so that it becomes easier to interpret. Specifically, a better way to visualize use and reuse should be devised, instead of arrows on top of the source code. Furthermore, the programmer would be helped further if also the code that is executed between use and long distance-reuse could be highlighted. Another extension might be to indicate the loops that carry the long distance reuses.

Appendix A

Computed Forward Reuse Distances: Examples

In this appendix, the computed reuse distances for a number of programs are shown. The iteration space of each reference is partitioned, where each partition has an Ehrhart polynomial describing the forward reuse distance. The dominant domains and dominant polynomials are indicated in bold.

A.1 Cholesky

The Cholesky factorization is shown in figure. The calculated forward reuse distances are shown below:

```
DO J=1,N
  DO L=J,N
    DO K=1,J-1
      A(L+0,J) = A(L-0,J) - A(L,K) * A(J,K)
    ENDDO
  ENDDO
  A(J+0,J) = SQRT(A(J-0,J)+1)
  DO M=J+1,N
    A(M,J+0) = A(M,J-0) / A(J,J+0)
  ENDDO
ENDDO
```

Figure A.1: Cholesky factorization

- Reference #39 `a(l+0,j)`

$$\begin{cases} \mathbf{0} & \textbf{in domain } \mathbf{k+2 \le j \le l \le n \wedge 1 \le k} \\ -1-kl+kn+n & \text{in domain } 1+k=j \wedge 2 \le j \le l-2 \wedge l < n \\ -1+l+ln-l^2 & \text{in domain } 1+k=j \wedge l=j \wedge 2 \le j < n \\ -1-k+l & \text{in domain } 1+k=j \wedge n=l \wedge 2 \le j \le l-2 \\ 1+k-kl+kn-l+n & \text{in domain } 1+k=j \wedge l=1+j \wedge 2 \le j \le n-2 \\ 1 & \text{in domain } k+2=j+1=l=n \wedge 2 \le j \\ \infty & \text{in domain } 1+k=j \wedge l=j \wedge n=j \wedge 2 \le j \end{cases} \tag{A.1}$$

- Reference #44 `a(l-0,j)`

$$\begin{cases} \mathbf{2} & \textbf{in domain } \mathbf{1 \le k < j < l \le n} \\ 1 & \text{in domain } l=j \wedge 1 \le k < j \le n \end{cases} \tag{A.2}$$

- Reference #49 `a(l,k)`

$$\begin{cases} \mathbf{-1+jn-j^2+l} & \textbf{in domain } \mathbf{k+1, 3 \le j \le l-2 \wedge l < n \wedge 1 \le k} \\ 2j-jl+jn & \text{in domain } 2 \le k+1, 3 \le j=l-1 \le n-2 \\ 0 & \text{in domain } l=j \wedge 1 \le k < j \le n \\ jl-j^2-k+n & \text{in domain } n=l \wedge k+2 \le j \le l-2 \wedge 1 \le k \\ 1+2j-k-l+n & \text{in domain } l=1+j \wedge n=1+j \wedge 1 \le k \le j-2 \\ 2-2j+jl-j^2+k+n & \text{in domain } 1+k=j \wedge n=l \wedge 3 \le j \le l-2 \\ -5+l+2n & \text{in domain } j=2 \wedge k=1 \wedge 3 \le l < n \\ -5+2l+n & \text{in domain } l=n \wedge k=1 \wedge j=2 \wedge 4 \le n \\ 3+k-l+n & \text{in domain } k+2=j+1=l=n \wedge 3 \le j \\ 4 & \text{in domain } j=2 \wedge k=1 \wedge l=3 \wedge n=3 \end{cases} \tag{A.3}$$

- Reference #52 `a(j,k)`

$$\begin{cases} \mathbf{-1+2j} & \textbf{in domain } \mathbf{k+2 \le j < l < n \wedge 1 \le k} \\ 1+2k & \text{in domain } 1+k=j \wedge 3 \le j \le l < n \\ j+k & \text{in domain } l=j \wedge k+2 \le j < n \wedge 1 \le k \\ \infty & \text{in domain } n=l \wedge 1 \le k < j \le l \\ 3 & \text{in domain } j=2 \wedge k=1 \wedge 2 \le l < n \end{cases} \tag{A.4}$$

- Reference #60 `a(j+0,j)`

$$\begin{cases} \mathbf{1} & \textbf{in domain } \mathbf{1 \le j < n} \\ \infty & \text{in domain } n=j \wedge 1 \le j \end{cases} \tag{A.5}$$

- Reference #65 `a(j-0,j)`

$$\begin{cases} \mathbf{0} & \textbf{in domain } \mathbf{1 \le j \le n} \end{cases} \tag{A.6}$$

- Reference #80 `a(m,j+0)`

$$\begin{cases} -\mathbf{j} + \mathbf{jm} - \mathbf{j^2} + \mathbf{n} & \textbf{in domain } \mathbf{2 \le j \le m - 2 \wedge m < n} \\ 1 + j - m + n & \text{in domain } m = 1 + j \wedge 2 \le j \le n - 2 \\ -1 - j + jm - j^2 + m & \text{in domain } n = m \wedge 1 \le j \le m - 2 \\ -2 + m + n & \text{in domain } j = 1 \wedge 2 \le m < n \\ j & \text{in domain } m = 1 + j \wedge n = 1 + j \wedge 2 \le j \\ 1 & \text{in domain } j = 1 \wedge m = 2 \wedge n = 2 \end{cases} \tag{A.7}$$

- Reference #85 `a(m,j-0)`

$$\begin{cases} \mathbf{1} & \textbf{in domain } \mathbf{1 \le j < m \le n} \end{cases} \tag{A.8}$$

- Reference #90 `a(j,j+0)`

$$\begin{cases} \mathbf{2} & \textbf{in domain } \mathbf{1 \le j < m < n} \\ \infty & \text{in domain } n = m \wedge 1 \le j < m \end{cases} \tag{A.9}$$

```
      DO 100 G = 1, N
        DO 100 H = 1, L
100           C(G,H) = 0.
      DO  J = 1, M
        DO  K = 1, N
          DO  I = 1, L
            C(I,K) = C(I,K+0) + A(I,J) * B(J,K)
          ENDDO
        ENDDO
      ENDDO
```

Figure A.2: matrix multiplication

A.2 Matrix Multiplication

The matrix multiplication code is shown in figure A.2. The calculated forward reuse distances are shown below:

- Reference #837 `c(g,h)`

FRD	in domain
$\mathbf{L(h-g+N+1)}$ $\mathbf{+N(1-h)+gh-1}$	$\mathbf{2 \le g < N \le L}$ $\mathbf{\wedge 2 \le h \le N, L-1 \wedge 1 \le M}$
$\mathbf{-1+gh-gL+2L+LN}$	$\mathbf{2 \le g \le L < N \wedge 2 \le h < L \wedge 1 \le M}$
$-2+h+hL-hN+LN+N$	$g = 1 \wedge 2 \le h \le N \le L \wedge h < L \wedge 1 \le M$
$-1+g+hL+L$	$N = g \wedge 2 \le h \le g \le L \wedge h < L \wedge 1 \le M$
∞	$1 \le g \le N < h \le L$
∞	$1 \le g \le N \wedge 1 \le h \le L \wedge M \le 0$
∞	$1 \le h \le L < g \le N$
$-1+gh-gL+h+L+LN$	$L = h \wedge 2 \le g \le h < N \wedge 1 \le M$
$-2+2g-gL+L+LN$	$h = 1 \wedge 2 \le g \le L, N-1 \wedge 1 \le M$
$-2+h+L+LN$	$g = 1 \wedge 2 \le h < L < N \wedge 1 \le M$
$g(h-L)+h(1+L-N)$ $+N(L+1)-1$	$N = h \wedge L = h \wedge 2 \le g < h \wedge 1 \le M$
$-2+2h+LN$	$L = h \wedge g = 1 \wedge 2 \le h < N \wedge 1 \le M$
$-1+LN$	$g = 1 \wedge h = 1 \wedge 1 \le M \wedge 2 \le N \wedge 1 \le L$
$-2+2g+L$	$N = g \wedge h = 1 \wedge 2 \le g \le L \wedge 1 \le M$
$-2+2h+hL-hN$ $-L+LN+N$	$h = N \wedge L = N \wedge g = 1 \wedge 2 \le N \wedge 1 \le M$
$-1+g+h+hL$	$h = g \wedge N = g \wedge L = g \wedge 2 \le g \wedge 1 \le M$
$-1+L$	$g = 1 \wedge h = 1 \wedge N = 1 \wedge 1 \le M \wedge 2 \le L$
∞	$g = 1 \wedge h = 1 \wedge L = 1 \wedge N = 1$

- Reference #859 `c(i,k)`

FRD	in domain
$\mathbf{2L+LN+N}$	$\mathbf{2 \le i < L \wedge 1 \le j < M \wedge 2 \le k < N}$
$-1+2i+LN+N$	$L = i \wedge 1 \le j < M \wedge 2 \le k < N \wedge 2 \le i$
$-i+k+kL+2L$	$N = k \wedge 2 \le i < L \wedge 1 \le j < M \wedge 2 \le k$
$-1+i+L+LN+N$	$k = 1 \wedge 2 \le i < L \wedge 1 \le j < M \wedge 2 \le N$
$-1+2L+LN+N$	$i = 1 \wedge 1 \le j < M \wedge 2 \le k < N \wedge 2 \le L$
∞	$M = j \wedge 1 \le i \le L \wedge 1 \le k \le N \wedge 1 \le j$
$2L$	$k = 1 \wedge N = 1 \wedge 2 \le i < L \wedge 1 \le j < M$
$-2+k+kL+2L$	$N = k \wedge i = 1 \wedge 1 \le j < M \wedge 2 \le k \wedge 2 \le L$
$-1+L+LN+N$	$i = 1 \wedge k = 1 \wedge 1 \le j < M \wedge 2 \le L \wedge 2 \le N$
$2N$	$i = 1 \wedge L = 1 \wedge 1 \le j < M \wedge 2 \le k < N$
$-2+3i-L+LN+N$	$L = i \wedge k = 1 \wedge 1 \le j < M \wedge 2 \le i \wedge 2 \le N$
$-1+i+k+kL$	$L = i \wedge N = k \wedge 1 \le j < M \wedge 2 \le i \wedge 2 \le k$
$-1+2i$	$L = i \wedge k = 1 \wedge N = 1 \wedge 1 \le j < M \wedge 2 \le i$
$-1+2N$	$i = 1 \wedge k = 1 \wedge L = 1 \wedge 1 \le j < M \wedge 2 \le N$
$-1+2k$	$k = N \wedge i = 1 \wedge L = 1 \wedge 1 \le j < M \wedge 2 \le N$
$-1+2L$	$i = 1 \wedge k = 1 \wedge N = 1 \wedge 1 \le j < M \wedge 2 \le L$
∞	$i = 1 \wedge k = 1 \wedge L = 1 \wedge N = 1 \wedge 1 \le j \le M$

- Reference #862 `c(i,k+0)`

FRD	in domain
$\mathbf{2}$	$\mathbf{1 \le i \le L \wedge 1 \le j \le M \wedge 1 \le k \le N}$

- Reference #867 `a(i,j)`

FRD	in domain
$\mathbf{2+2L}$	$\mathbf{2 \le i \le L \wedge 1 \le j \le M \wedge 1 \le k < N}$
$1+2L$	$i = 1 \wedge 1 \le j \le M \wedge 1 \le k < N \wedge 2 \le L$
∞	$N = k \wedge 1 \le i \le L \wedge 1 \le j \le M \wedge 1 \le k$
3	$i = 1 \wedge L = 1 \wedge 1 \le j \le M \wedge 1 \le k < N$

- Reference #870 `b(j,k)`

FRD	in domain
$\mathbf{3}$	$\mathbf{1 \le i < L \wedge 1 \le j \le M \wedge 1 \le k \le N}$
∞	$L = i \wedge 1 \le j \le M \wedge 1 \le k \le N \wedge 1 \le i$

```
do i = 1,n
  do j = 1,n
    if(j.ne.i) then
      f=a(j,i)/a(i,i)
      do k=i+1,n+1
        a(j,k)=a(j,k)-f*a(i,k)
      enddo
    endif
  enddo
enddo

! substitution
do l=1,n
  x(l)=a(l,n+1)/a(l,l)
enddo

! backsubstitution
do g=1,n
  c(g) = 0
    do h=1,n
      c(g) = c(g) + b(g,h)*x(h)
    enddo
  delta = delta + (c(g)-b(g,n+1))**2
enddo
```

Figure A.3: Gauss-Jordan elimination

A.3 Gauss-Jordan

The Gauss-Jordan elimination is shown in figure A.3. The calculated forward reuse distances are shown below:

- Reference #297 `b(i,j)`

FRD	in domain
∞	$\mathbf{1 \leq i \leq n \wedge 1 \leq j \leq n+1}$

- Reference #300 `a(i,j)`

FRD	in domain
∞	$\mathbf{1 \leq i \leq n \wedge 1 \leq j \leq n+1}$

- Reference #327 `a(j,i)`

FRD	in domain
∞	$\mathbf{1 \leq j < i \leq n}$
∞	$\mathbf{1 \leq i < j \leq n}$

- Reference #330 `a(i,i)`

FRD	in domain
$\mathbf{3 - 2i + 2n}$	$\mathbf{j + 2 \leq i \leq n \wedge 1 \leq j}$
$\mathbf{3 - 2i + 2n}$	$\mathbf{1 \leq i < j < n}$
$-2 + 2i - in + n + n^2$	$j = n \wedge 1 \leq i \leq n - 2$
$1 - 2j + 2n$	$1 + j = i \wedge 2 \leq i < n$
$-2 + 3n$	$i = n \wedge 1 + j = n \wedge 2 \leq n$
$-2 + \frac{3i}{2} \frac{-i^2}{2} + \frac{3n}{2} + \frac{n^2}{2}$	$1 + i = n \wedge j = n \wedge 3 \leq n$
4	$i = 1 \wedge j = 2 \wedge n = 2$

- Reference #344 `a(j,k)`

FRD	in domain
$-\mathbf{in}-\mathbf{j}+\mathbf{2n}+\mathbf{n^2}$	$\mathbf{1 \le i < k \le n+1 \wedge i+3 \le j < n}$
$-1-in-j+2n+n^2$	$j+2 \le i < k \le n \wedge 1 \le j$
$2-i+ij-i^2$ $-2j-jn+4n+n^2$	$1 \le i = j-2 < k \le n+1 \wedge i \le n-3$
$-1-ij+j+jn$	$j = n \wedge 1 \le i < k \le n \wedge i \le n-3$
$-2+n+n^2$	$i = 1 \wedge j = 2 \wedge 2 \le k \le n+1 \wedge 3 \le n$
$-2-in-j+k+n+n^2$	$k = 1+n \wedge j+2 \le i \le n-2 \wedge 1 \le j$
$1-3i-ij-2in+i^2$ $+j+jn+3n+n^2$	$1+j = i \wedge 3 \le i < k \le n$
$5-6j-2jn+j^2+k+5n+n^2$	$j = 1+i \wedge 2 \le i \le k-3 \wedge k \le n$
$-2+n^2$	$i = 2 \wedge j = 1 \wedge 4 \le k \le n+1 \wedge 4 \le n$
$-1+3n$	$2+i = n \wedge j = n \wedge 3, k \le n \le k+1$
$5-5k-2kn+k^2+5n+n^2$	$j = 1+i \wedge k = 1+i \wedge 2 \le i \le n-2$
$-3i-ij-2in+i^2$ $+j+jn+k+2n+n^2$	$1+j = i \wedge k = 1+n \wedge 3 \le i \le n-2$
$4-6j-2jn+j^2+2k+4n+n^2$	$j = 1+i \wedge k = 1+n \wedge 2 \le i \le n-2$
$-3+j+2n$	$i = n \wedge k = 1+n \wedge 1 \le j \le n-2$
$i-in-j+n+n^2$	$1+i = n \wedge k = 1+n \wedge 1 \le j \le n-3$
$-2-ij+j+jn+k-n$	$j = n \wedge k = 1+n \wedge 1 \le i \le n-3$
$7-4j-2jn+j^2-k+5n+n^2$	$j = 1+i \wedge k = 2+i \wedge 2 \le i \le n-2$
$-3+3j$	$i = n \wedge 1+j = n \wedge k = 1+n \wedge 3 \le n$
$2-2i-ij-2in+i^2$ $+j+jn+2n+n^2$	$4 \le 1+i = n = 2+j = k-1$
$-i+in-i^2+k+n$	$2+i = n \wedge j = n \wedge k = 1+n \wedge 3 \le n$
3	$1+i = n \wedge j = n \wedge k = 1+n \wedge 3 \le n$
$-2+n^2$	$i = 2 \wedge j = 1 \wedge k = 3 \wedge 3 \le n$
$3-2k+2n$	$1+i = n \wedge j = n \wedge k = n \wedge 3 \le n$
3	$i = 1 \wedge j = 2 \wedge n = 2 \wedge 2 \le k \le 3$
0	$i = 2 \wedge j = 1 \wedge k = 3 \wedge n = 2$
7	$i = 2 \wedge j = 1 \wedge k = 4 \wedge n = 3$

- Reference #347 `a(j,k)`

FRD	in domain
$\mathbf{1}$	$\mathbf{1 \le i < j \le n \wedge i < k \le n+1}$
1	$1 \le j < i < k \le n+1$

- Reference #351 `a(i,k)`

FRD	in domain
$\mathbf{4 - 2i + 2n}$	$\mathbf{1 \leq i < j < n \wedge i < k \leq n + 1}$
$4 - 2i + 2n$	$1 \leq j < i < k \leq n$
$-1 + 2n$	$j = n \wedge i = 1 \wedge 2 \leq k \leq n + 1 \wedge 3 \leq n$
$2 - 2i + 2k$	$k = 1 + n \wedge 1 \leq j < i < n$
$2 + in - i^2 - k + 2n$	$j = n \wedge 2 \leq i \leq k - 3 \wedge k \leq n$
4	$i = n \wedge k = 1 + n \wedge 1 \leq j \leq n - 2$
$in - i^2 + k$	$j = n \wedge k = 1 + n \wedge 2 \leq i \leq n - 2$
$-1 + 2n$	$1 + i = n \wedge j = n \wedge 3, k - 1 \leq n \leq k$
$1 + k + kn - k^2 + n$	$k = 1 + i \wedge j = n \wedge 2 \leq i \leq n - 2$
$4 + i + in - i^2 - 2k + 2n$	$k = 2 + i \wedge j = n \wedge 2 \leq i \leq n - 2$
$-3 + 3n$	$i = n \wedge 1 + j = n \wedge k = 1 + n \wedge 2 \leq n$
3	$i = 1 \wedge j = 2 \wedge n = 2 \wedge 2 \leq k \leq 3$

- Reference #366 `x(l)`

FRD	in domain
$\mathbf{-l + 3n}$	$\mathbf{1 \leq l < n}$
$2l$	$l = n \wedge 2 \leq n$
2	$l = 1 \wedge n = 1$

- Reference #368 `a(l,n+1)`

FRD	in domain
∞	$\mathbf{1 \leq l \leq n}$

- Reference #373 `a(l,l)`

FRD	in domain
∞	$\mathbf{1 \leq l \leq n}$

- Reference #384 `c(g)`

FRD	in domain
$\mathbf{0}$	$\mathbf{1 \leq g \leq n}$

- Reference #393 `c(g)`

FRD	in domain
$\mathbf{0}$	$\mathbf{1 \leq g \leq n \wedge 1 \leq h \leq n}$

- Reference #395 `c(g)`

FRD	in domain
$\mathbf{2}$	$\mathbf{1 \leq g \leq n \wedge 1 \leq h \leq n}$

- Reference #397 `b(g,h)`

FRD	in domain
∞	$\mathbf{1 \leq g \leq n \wedge 1 \leq h \leq n}$

- Reference #400 `x(h)`

FRD	in domain
$\mathbf{2 + 2n}$	$\mathbf{1 \leq g < n \wedge 1 \leq h \leq n}$
∞	$g = n \wedge 1 \leq h \leq n$

- Reference #408 `c(g)`

FRD	in domain
∞	$\mathbf{1} \leq \mathbf{g} \leq \mathbf{n}$

- Reference #410 `b(g,n+1)`

FRD	in domain
∞	$\mathbf{1} \leq \mathbf{g} \leq \mathbf{n}$

Index

k-polyhedron, 111
q-periodic number, **134**
`.nt1`, **177**
`.nt2`, **177**
`.nta`, **177**
`.t1`, **177**
3C's model, 72

accessed data set, **92**, 118, 122, 123
affine hull, **111**
algorithmic optimizations, 62
Alpha, 221
array layout, 71
array padding, 71
array transposition, 212
Art, 216
associativity, **75**
Athlon, 221
Atlas, 61
Atlas 2, 61

backward reuse, 123
backward reuse distance, **92**, 125
backward reuse distance distribution, **93**

cache, 61, 62
cache bypass, 86
cache equations, 127, 170
cache eviction, 70
cache hint, 72, **176**
cache hints, 73, 81
cache miss equations, 170
cache pollution, 86
cache remapping, 69
cache set, 77
cache sets, **75**
cache shadow, 80
cache size, **75**
cache stay, 70
cache-centric visualization, 210
capacity miss, 62, **95**
capacity misses, 70
CEC-201, 61
Cholesky factorization, 127
Clauss's theorem, 135
CME, 170
cold miss, 62, **95**
compiler optimizations, 62
cone
    generator, 110
conflict miss, 62, 71, **95**
convex polyhedron, **108**
copying, 84, 86
core memory, 61
Cprof, 210
cumulative reuse distance distribution, 127
CVT, 210

dénombrant, 133
data layout, 70
data dependency, 79, 81
data layout, 71

data tile, **73**
degenerate domain, **138**
denominator of a polyhedron, **113**
denominator of a vertex, **113**
dimension of a polyhedron, **111**
Dinero, 84
disjunctive normal form, 118
division constraint, 120
DNF, 118
dominant domain, **168**
dominant polynomial, **168**
drum memory, 61
dynamic cache hint, 182
dynamic hints, **179**

Ehrhart, 132
Ehrhart polynomial, 122
Ehrhart polynomial, 125, **135**
Ehrhart's theorem, 135
enumerating
    non-parameterized polytope, 137
enumerator, 133
EPIC, **176**
Equake, 218
ETL Mk-6, 61

face of a polyhedron, **112**
face-lattice, **112**
field reordering, 212
field reorganization, 70
formal power series, 143
forward reuse, 123
forward reuse distance, **92**, 125
forward reuse distance distribution, **93**
free variable, 113
fusion, 70

generating function, 143
generator, **110**

hardware optimizations, 62
hash function, 71
homothetic polyhedron, **133**
HPCView, 210
human effort, 211

IA-64, 177
IBM/360, 62
implicit equality, **111**
integer polyhedron, **108**
interpolation
    limitations, 137
interpolation, computing enumerator using, 136
interweaving threads, 81
Itanium, 62, 221
iteration space, 119
iteration tile, **73**

large periods, 141
latency hiding, 69
lexicographical ordering, 119
line, **109**
line size, **75**
locality, 70
loop coalescing, 82
loop fusion, 70, 212
loop tiling, 70, 73, 212
loop transformation, 70, 71
LRU replacement policy, **94**
LRW, 84

machine description, 187
Mcf, 214, 215
MDES, **187**
memory access, **91**, 122
memory hierarchy, 61
memory line, **75**, **91**
memory line utilization, 213, **213**

memory location, 119
memory reference, **91**, 122
Memspy, 210
microarchitectural optimizations, 62
Minkowski representation, 109
modulo constraint, 120
Mtool, 210
multithreading, 72

non-parameterized polytope, 137

Olden benchmark, 192
Omega library, 118
Open64, **187**
ORC compiler, 213
overhead, 214

padding, 84, 86
paging, 61
parameter, 113
parameterized polyhedron, **113**
periodic number, **134**
platform-independent optimizations, 211
PME, 170
Polyèdres homothétiques, 132
polyhedral cone, **110**
polyhedral model, 108, 122
polyhedral program model, 118
polyhedron
  k-polyhedron, 111
  convex, 108
  integer, 108
Polylib, 137
polynôme arithmétique, 133
polytope, **110**
prefetch, 72
prefetching, 212
preload, 72, 176
Presburger formula, 118
Presburger formula, 108, 117, **118**, 122
Pro64, **187**
probabilistic miss equations, 170
process thread, 79
processing thread, 69, 77
program-centric visualization, 210
pseudo-period, **135**
pseudo-polynomial, **135**

ray, 109, **109**
reference distance, 74
references, 122
register pressure, 179, 191
remap thread, 69
remap thread, 77, 79
replacement policy, 69, 72
reuse distance, **92**, 118, 125
reuse distance distribution, **93**, 213
reuse pair, **92**, 122
reuse pairs, 122
Rivet, 210

SIP, 210
skewed-associative cache, 71
software pipelining, 191, 192
source cache hint, 176
source cache specifier, **176**
spatial locality, 212
spatial locality, 69, 70, 73, 212
SPEC2000, 62, 64, 215
Spec95FP, 192
static cache hint, 179
static hints, **179**
supporting cone, **110**

target cache hint, 176

target cache specifier, **176**
temporal locality, 69, 70, 73, 212
thread interweaving, 81
thread scheduling, 81, 82
thread synchronization, 79
tile set, **73**, 77
tile size, 81
tile size selection, 81, 86
tiled loops, **73**
tiling, 70
tiling loops, **73**
Trimaran, 84

unimodular cone, **110**, 145
unimodular generator, **110**

validity domain, **117**
    partitioning, 117
VCG, 215
vertex, 109, **109**
victim cache, 71

working set, 71

XML, 214
XSLT, 214

Bibliography

[1] S. Abraham and B. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett-Packard Company, nov 1994. 4.6.1

[2] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO'93)*, volume 24 of *SIGMICRO Newsletter*, pages 139–152, Los Alamitos, CA, USA, Dec. 1993. IEEE Computer Society Press. 5.1.1

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. 5.3.2

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. 4.7

[5] G. Almàsi, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the workshop on Memory System Performance*, pages 37–43. ACM Press, 2002. 3.4

[6] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the fourth International Conference on Parallel and Distributed Information Systems*, pages 92–107. IEEE Computer Society, 1996. 3.4

[7] G. Alverson, S. Kahan, R. Korry, and C. McCann. Scheduling on the Tera MTA. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 19–44, 1994. 2.1.3, 3.3.1

[8] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings*

of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pages 166–178. ACM Press, 1995. 2.1.1

[9] C. Aragon and R. Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science*, pages 540–545, 1989. 3.2.1

[10] D. F. Bacon, J.-H. Chow, D. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*, pages 270–282, Toronto, Canada, Oct. 1994. 2.1.2

[11] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, Dec. 1994. 3.3.2

[12] J.-L. Baer. 2k papers on caches by Y2K: Do we need more? Keynote address at the 6th International symposium on High-Performance Computer Architecture, January 2000. 1.1

[13] A. Barvinok and J. Pommersheim. An algorithmic theory of lattice points in polyhedra. *New Perspectives in Algebraic Combinatorics*, (38):91–147, 1999. 4.1.1, 4.4.1, 4.4.1, 4.4.1

[14] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In *34th Annual Symposium on Foundations of Computer Science*, pages 566–572. IEEE, Nov. 1993. 4.3.3, 4.4.1

[15] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubjana, october 2003. 4.1.4, 4.6

[16] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 5

[17] B. Bennett and V. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, pages 353–357, 1975. 3.4

[18] E. Berg and E. Hagersten. SIP: Performance tuning through source code interdependence. In *Proceedings of the 8th International Euro-Par Conference*, pages 177–186, 2002. 6.1, 6.1, 6.4

[19] K. Beyls. Automatische parallellisatie van Java-programma's. Master's thesis, Ghent University, 1999. 4.6

[20] K. Beyls and E. D'Hollander. Cache remapping to improve the performance of tiled algorithms. In *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Notes in Computer Science, pages 998–1007, 2000. 1.2, 7.1

[21] K. Beyls and E. D'Hollander. Compiler generated multithreading to alleviate memory latency. *Journal of Universal Computer Science, special issue on Multithreaded Processors and Chip-Multiprocessors*, 6(10):968–993, oct 2000. 1.2, 5.2, 7.1

[22] K. Beyls and E. D'Hollander. Compile-time cache hint generation for EPIC architectures. In *Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, 2002. 1.2, 7.1

[23] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, pages 265–274, 2002. 1.2, 7.1

[24] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 2004. in submission. 7.1

[25] K. Beyls and E. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *Proceedings of the International Conference on Computational Science*, volume 3038 of *Lecture Notes in Computer Science*, pages 448–455, June 2004. 1.1, 1.2, 7.1

[26] K. Beyls, E. D'Hollander, and Y. Yu. JPT: A Java parallelization tool. In J. J. Dongarra, E. Luque, and T. Margalef, editors, *6th European PVM/MPI Users' Group Meeting, proceedings*, volume 1697 of *Lecture Notes in Computer Science*, pages 173–180. Springer-Verlag, 1999. 4.6

[27] K. Beyls, E. D'Hollander, and Y. Yu. Visualization enables the programmer to reduce cache misses. In *Conference on Parallel and Distributed Computing and Systems*, 2002. 1.2, 7.1

[28] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, Aug 2001. 1.2, 7.1

[29] B. Boigelot and L. Latour. Counting the solutions of Presburger equations without enumerating them. In *Proc. 6th International Conference on Implementations and Applications of Automata (Revised Papers), volume 2494, Lecture Notes in Computer Science*, pages 40–51, July 2001. 4.5

[30] B. Boigelot and L. Latour. Counting the solutions of Presburger equations without enumerating them. *Theoretical Computer Science*, (313):17–29, 2004. 4.5

[31] R. Bosch and C. S. et al. Rivet: A flexible environment for computer systems visualization. *Computer Graphics-US*, 34(1):68–73, Feb. 2000. 6.1, 6.1, 6.4

[32] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994. 2.1.1

[33] M. Brehob and R. J. Enbody. An analytic model of locality and caching. Technical Report MSU-CSE-99-31, Michigan State University, August 1999. 3.4

[34] M. Brion and M. Vergne. Residue formulae, vector partition functions and lattice points in rational polytopes. *J. Amer. Math. Soc.*, 10:797–833, 1997. 4.4.1

[35] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988. 1

[36] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM Press, 1991. 2.1.3

[37] J. F. Cantin and M. D. Hill. Cache performance for selected SPEC CPU2000 benchmarks. *Computer Architecture News (CAN)*, September 2001. 1.2, 1.2, 3.3.1

[38] L. Carter, J. Feo, and A. Snavely. Performance and programming experience on the Tera MTA. In *SIAM Conference on Parallel Processing*, March 1999. 2.1.3

[39] G. C. Caşcaval. *Compile-time performance prediction of scientific programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000. 3.4, 4.7, 4.7

[40] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998. 2.4

[41] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 47(1):25–33, Feb. 1996. 3.3.1

[42] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th International Symposium on Computer Architecture*, may 1999. 5.5

[43] S. Chatterjee, V. Jain, A. R. Lebeck, S. Mundhra, and M. S. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *13th ACM International Conference on Supercomputing (ICS '99)*, June 1999. 2.1.1, 2.1.2

[44] S. Chatterjee, E. Parker, P. Hanlon, and A.R.Lebeck. Exact analysis of the cache behavior of nested loops. In *Conference on Programming Languages Design and Implementation*, pages 286–297, 2001. 4.7, 4.7

[45] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 13–24, Atlanta, May 1999. ACM Press. 2.1.1

[46] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1–12, May 1999. 2.1.1

[47] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002. 2.1.3, 5.5

[48] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–285. ACM, May 1996. 4.1.4

[49] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. In *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*. IEEE Computer Society, Aug. 1996. 4.3.2, 4.3.2

[50] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, 1998. 4.3, 22, 23, 4.1.2, 4.3.1, 5, 4.6

[51] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice Hall, Englewood Cliffs, 1973. 3.1.3, 3.4, 4.1.4

[52] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN'95: conference on programming language design and implementation*, pages 279–290, June 1995. 2.1.2, 2.4

[53] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, July 1995. 4.1.4

[54] I. Corporation. Vtune: a visual tuning environment. http: //-support.intel.com/support/performancetools/vtune/. 6.1, 6.4

[55] A. Darte. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 149–157, Newport Beach, California, Oct. 12–16, 1999. IEEE Computer Society Press. 2.1.1

[56] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. In *CASES*, pages 298–308. ACM, 2003. 4.1.4

[57] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes, Mar. 2003. http://www.math.ucdavis.edu/˜ latte/theory.html. 4.4.1, 4.4.1, 4.4.1, 4.4.1, 4.4.1, 4.4.2

[58] E. D'Hollander, F. Zhang, and Q. Wang. The Fortran parallel transformer and its programming environment. *Journal of Information Science*, 106:293–317, 7 1998. 4.6

[59] C. Ding. *Improving Effective Bandwidth through compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, 2000. 3.4

[60] C. Ding and K. Kennedy. Improving cache performance of dynamic applications through data and computation reorganization at run time. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 229–241. ACM Press, May 1999. 3.4

[61] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64:108–134, 2004. 2.1.1

[62] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Conference on Programming Languages Design and Implementation'03*. ACM, 2003. 3.4, 3.4, 4.7

[63] C. C. Douglas and J. H. et al. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal.*, 10:25–40, 2000. 6.3.3

[64] S. J. Eggers, H. M. Levy, J. L. Lo, J. S. Emer, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, sept/oct 1997. 2.1.3

[65] E. Ehrhart. *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire*. International Series of Numerical Mathematics. Birkhäuser Verlag, 1977. 4.3, 4.3.1, 4.3.1

[66] P. Feautrier. Pip-software. http://www.prism.uvsq.fr/p̃af/. 4.6

[67] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, Sept. 1988. 4.6

[68] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, Oct. 1992. 4.1.4

[69] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, Dec. 1992. 4.1.4

[70] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996. 4.1

[71] B. Fraguela, R. Doallo, J. Touriño, and E. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Computing*, 2004(30):225–248, 2004. 4.7, 4.7

[72] B. B. Fraguela, R. Doallo, and E. L. Zapata. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Trans. Comput.*, 52(3):321–336, 2003. 4.7, 4.7

[73] F. H. M. Franssen and F. Balasa. Modeling multidimensional data and control flow. *IEEE Transactions on Very Large Scale Integration Systems*, 1(3):319–326, Sept. 1993. 4.1.4

[74] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996. 3.1

[75] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour*. PhD thesis, Princeton University, November 1999. 2.2.2, 3.1.1, 4, 4.7, 4.7

[76] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *International Conference on Supercomputing*, pages 317–324, 1997. 4.7

[77] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999. 4.7

[78] A. Goldberg and J. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, 1993. 6.1, 6.1, 6.4

[79] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989. 4.4.2

[80] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Trans. Comput.*, 45(11):1319–1326, 1996. 3.4

[81] P. Grun, N. Dutt, and A. Nicolau. MIST: An algorithm for memory miss traffic management. In *International Conference on Computer Aided Design*, pages 431–437, 2000. 5.5

[82] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transaction on Computers*, 48(10):1009–1024, oct 1999. 4.7, 4.7

[83] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2002. 1, 1.1, 1, 1.1, 3.3.1

[84] M. Hill and J. Elder. DineroIV tracedriven uniprocessor cache simulator. http://www.cs.wisc.edu/markhill/DineroIV, 1998. 2.3.2

[85] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989. 3.1.3, 3.3.1, 3.4

[86] *IA-64 Application Developer's Architecture Guide*, May 1999. 2.2.5, 5.1, 5.2, 2

[87] *Intel Itanium 2 Processor Reference Manual*, June 2002. 5.1.2, 5.1

[88] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88. Proceedings of the conference on Principles of Programming Languages*, pages 319–329. ACM Press, 1988. 2.1.1, 3.3.2

[89] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted replacement mechanisms for embedded systems. In *International Conference on Computer Aided Design*, pages 119–126, nov 2001. 2.1.4, 5.2.1, 5.4.4, 5.5

[90] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Comput.*, 48(12):1338–1354, 1999. 2.1.4

[91] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, pages 364–373, May 1990. 2.1.2, 3.3.1

[92] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, 1999. 2.1.1

[93] G. Kane. *PA-RISC 2.0 architecture*. Prentice Hall, 1996. 2.2.5

[94] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1998. 3

[95] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL_PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard, February 2000. 5.1

[96] T. Kelly and D. Reeves. Optimal web cache sizing: Scalable methods for exact solutions. *Computer Communications*, 24(2):163–173, 2001. 3.4

[97] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical report, 1995. 4.5, 4.6

[98] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *In Proceedings of the IEEE First International Conference on Algorithms And Architectures for Parallel Processing*, Bris-bane, Australia, April 1995. 2.1.1

[99] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, OR, 1993. Berlin: Springer Verlag. 2.1.1

[100] T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner. One-level storage system. *IRE Trans. on Electronic Computers*, pages 223–235, April 1962. 1.1

[101] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *ACM SIGMETRICS conference*, pages 212–213, 1991. 3.2, 3.4

[102] D. E. Knuth. *The Art of Computer Programming, Vol 3, Sorting and Searching*. Addison-Wesley, Reading, USA, 2 edition, 1998. 3.2.1

[103] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Conference on Programming Languages Design and Implementation*, pages 346–357, 1997. 2.1.1, 3.3.2

[104] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *International Conference on Supercomputing*, 1999. 3.3.2

[105] D. C. Kozen. *The Design and Analysis of Algorithms*. Text and Monographs in Computer Science. Springer-Verlag, 1992. 3.2.1, 3.2.1

[106] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991. 2.5, 2.8, 2.1.2, 2.13, 2.3.2, 2.4

[107] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994. 6.1, 6.1, 6.4

[108] K. Lee. The NAS860 library user's manual, 1993. 2.4

[109] E. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998. 2.1.1

[110] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *to appear in ACM Transactions on Programming Languages and Systems*, 2004. 2.1.1

[111] W.-F. Lin and S. K. Reinhardt. Predicting last-touch references under optimal replacement. Technical Report CSE-TR-447-02, University of Michigan, 2002. 2.1.4

[112] J. S. Liptay. Structural aspects of the system/360 model 85, part ii: The cache. *IBM Systems Journal*, 7(1):15–, 1968. 1.1

[113] V. Loechner. Polylib: A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France, Mar. 1999. 4.3.3

[114] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. Supercomput.*, 21(1):37–76, 2002. 2.1.1, 4.1.4

[115] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, Dec. 1997. 2, 22, 3, 4.1.2

[116] C.-K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer applications. *IEEE Transactions on Computers*, 48(2):134–141, February 1999. 2.1.3, 5.5

[117] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997. 2.1.1

[118] M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance in sequential and parallel programs. *IEEE Computer*, April 1995. 6.1, 6.1, 6.4

[119] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. 3.1.3, 3.4

[120] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996. 2.1.1, 3.3.2

[121] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999. 3.3.1, 3.3.2, 3.4

[122] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, Newport, Rhode Island, June 22–25, 1997. 2.1.1

[123] B. Meister. Using periodics in integer polyhedral problems. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France, Oct. 2003. 4.4.2

[124] J. Mellor-Crummey, R. Fowler, and G. Marin. HPCView: a tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–104, 2002. 6.1, 6.1, 6.4

[125] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of Computer Science, Stanford University, Mar. 1994. 4.7, 5.5

[126] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, Feb. 1998. 5.5

[127] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM SIGPLAN Notices*, 27(9):62–73, Sept. 1992. 2.1.3

[128] M. D. Nahas and W. A. Wulf. Data cache performance when vector-like accesses bypass the cache. Technical Report CS-97-16, Department of Computer Science, University of Virginia, July 5 1997. Mon, 16 Nov 1998 18:30:47 GMT. 2.1.4

[129] J. Ng, D. Kulkarni, W. Li, R. Cox, and S. Bobholz. Inter-procedural loop fusion, array contraction and rotation. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 114–124, Sept. 2003. 2.1.1

[130] F. Olken. Efficient methods for calculating the succes function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981. 3.2, 3.4

[131] Open64 compiler. http://sourceforge.net/projects/open64. 5.4.1

[132] Open research compiler. http://sourceforge.net/projects/ipf-orc. 5.4, 6.2.3

[133] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *MICRO'95*, pages 243–248, Ann Arbor, Michigan, Nov. 29–Dec. 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO. 5.5

[134] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE transactions on computers*, 48(2):142–149, Feb 1999. 2.5, 2.8, 2.13, 2.3.2, 2.4, 3.3.2

[135] E. Parker and S. Chatterjee. An automata-theoretic algorithm for counting solutions to Presburger formulas. In *13th International Conference on Compiler Construction*, pages 104–119, 2004. 4.5

[136] V. Pingali, S. McKee, W. Hseih, and J.B.Carter. Computation regrouping: Restructuring programs for temporal data cache locality. In *International Conference on Supercomputing*, pages 252–261, 2002. 6.4

[137] C. D. Polychronopoulos. Loop coalesing: A compiler transformation for parallel machines. In *International Conference on Parallel Processing*, pages 235–242. Pennsylvania State Univ. Press, Aug. 1987. 2.4.2, 2.2.5

[138] The polyhedral library. http://icps.u-strasbg.fr/PolyLib/. 4.3.3, 4.6

[139] M. Presburger. Über die Volständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 91–101, 1929. Warsaw, Poland. 4.1.3

[140] M. Prvulovic, D. Marinov, Z. Dimitrijevic, and V. Milutinovic. Split temporal/spatial caches: A survey and reevaluation of performance. *IEEE TCCA Newsletter*, pages 1–10, 1999. 2.1.4

[141] W. Pugh. Counting solutions to Presburger formulas: How and why. *ACM SIGPLAN Notices*, 29(6):121–134, jun 1994. 4.1.3, 4.1.4, 4.5, 4.5.1

[142] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, Feb. 1995. 4.1.3

[143] C. Pyo and G. Lee. Reference distance as a metric for data locality. In *HPC-ASIA 97*, pages 151–156, 1997. 2.2.1, 3.4

[144] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. In *ACM Transactions on Programming Languages and Systems (TOPLAS),Volume 22, Issue 5*, pages 773–815, Sept. 2000. 4.1.4

[145] B. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999. 5.4.1

[146] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, June 1998. 2.1.2, 3.3.2

[147] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998. 2.1.2

[148] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction (CC'99)*, March 1999. 2.4

[149] L. Robinson. Model 30-201 electronic digital computer. In *Proc. Joint AIEE-IRE Comput. Conf.*, pages 31–36, December 1951. 1.1

[150] S. Rosen. Electronic computers: A historical survey. *ACM Computing Reviews*, 1(1):7–36, 1969. 1.1

[151] G. Roth and K. Kennedy. Loop fusion in high performance Fortran. In *International Conference on Supercomputing*, pages 125–132, 1998. 2.1.1

[152] J. Sanchez and A. Gonzalez. A locality sensitive multi-module cache with explicit management. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 51–59, N.Y., June 20–25 1999. ACM Press. 2.1.4, 5.5

[153] G. Sander. Graph layout through the VCG tool. In *DIMACS International Workshop GD'94, Lecture Notes in Computer Science 894*, pages 194–205, 1995. 6.2.3

[154] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 264–273, New York, June 2–4 1997. ACM Press. 5.5

[155] M. S. Schlansker and B. R. R. Cover. EPIC: Explicitly parallel instruction computing. *IEEE Computer*, 33(2):37–45, Feb. 2000. 5.1

[156] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986. 12, 12, 2, 1, 15, 18

[157] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. ASPLOS VII*, Cambridge, MA, Octobe 1996. 2.4

[158] R. Seghir, S. Verdoolaege, K. Beyls, and V. Loechner. Analytical computation of Ehrhart polynomials and its applications in compile-time generated cache hints. Technical report, Université Louis Pasteur Strasbourg, 2004. 1.2, 7.1

[159] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, 2002. 2.4

[160] A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE '93*, Lecture Notes in Computer Science, pages 305–316, Munich, Germany, June 14–17, 1993. Springer-Verlag. 2.1.2, 3.3.1

[161] SGI Pro64 compiler. http://oss.sgi.com/projects/Pro64. 3.3.2

[162] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept./Oct. 2000. 5.1.2

[163] L.-P. Sheng. Implementatie van een C-interface voor de parallelliserende compiler FPT. Master's thesis, Ghent University, 1998. 4.6

[164] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99), Atlanta, Georgia, USA*, pages 215–228. ACM Press, 1999. 2.1.1

[165] The SPEC website. http://www.spec.org. 3.2.3, 6.3.1

[166] R. Sree, A. Settle, I. Bratt, and D. A. Connors. Compiler-directed resource management for active code regions. In *Proceedings of the 7th Workshop on Interaction between Compilers and Computer Architecture*, February 2003. 5.5

[167] R. A. Sugumar. *Multi-Configuration Simulation Algortihms for the Evaluation of Computer Architecture Designs*. PhD thesis, University of Michigan, August 1993. 3.4

[168] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *ACM Sigmetrics Conference*, pages 24–35, May 1993. 2.1.4, 3.3.1

[169] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comput. Syst.*, 13(1):32–56, 1995. 3.4

[170] E. Tam, J. Rivers, V. Srinivasan, G. Tyson, and E. Davidson. Active management of data caches by exploiting reuse information. *IEEE Transactions on Computers*, 1999. 2.1.4

[171] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings, Supercomputing '93*, pages 410–419, March 1993. 2.5, 2.8, 2.1.2, 2.13, 2.3.2, 2.4

[172] J. G. Thompson and A. J. Smith. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Trans. Comput. Syst.*, 7(1):78–117, 1989. 3.4

[173] Trimaran. *The Trimaran Compiler Research Infrastructure for Instruction Level Parallelism*. The Trimaran Consortium, 1998. http://www.trimaran.org. 2.5, 2.3.2

[174] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO'95*, pages 93–103, Ann Arbor, Michigan, Nov. 29–Dec. 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO. 2.1.4, 5.5

[175] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997. 3.4

[176] T. Ungerer, B. R. c;, and J. ˇ Silc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003. 2.1.3

[177] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *Proceedings of Languages, Compilers and Tools for Embedded Systems 2003, San Diego, California*. ACM, June 2003. 4.1.4

[178] E. Vanderdeijl, O. Temam, E. Granston, and G. Kanbier. The cache visualization tool. *IEEE Computer*, 30(7):71, 1997. 6.1, 6.1, 6.4

[179] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000. 5.5

[180] H. Vandierendonck. *Avoiding Mapping Conflicts in Microprocessors*. PhD thesis, Ghent University, 2003. 2.1.2

[181] X. Vera, J. LLosa, A. Gonzalez, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *Euro-Par*, pages 194–198, 2000. 4.7, 4.7

[182] X. Vera and J. Xue. Let's study whole-program cache behavior analytically. In *High-Performance Computer Architecture (HPCA'02)*, pages 175–186, Feb 2002. 4.7, 4.7

[183] S. Verdoolaege. The barvinok library. 4.6

[184] S. Verdoolaege, K. Beyls, M. Bruynooghe, R. Seghir, and V. Loechner. Analytical computation of Ehrhart polynomials and its applications for embedded systems. In *2nd Workshop on Optimizations for DSP and Embedded Systems*, Palo Alto, 3 2004. 1.2, 7.1

[185] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In D. Martin, editor, *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, The Hague, The Netherlands, June 2003. 2.1.1

[186] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems. Using the compiler to improve cache replacement decisions. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2002. 5.5

[187] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, dec 1993. 4.6

[188] M. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC14(2):270–271, April 1965. 1.1

[189] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, Chau-Wen, Tseng, M. W.

Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994. 3.2.3

[190] M. Wolf. Improving locality and parallelism in nested loops. Ph.d. thesis, Stanford University, 1992. 3.3.2

[191] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991. 2.1.1, 1, 3.3.2

[192] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286. IEEE Computer Society, 1996. 3.3.2

[193] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996. 3.3.2, 4.1.4

[194] M. J. Wolfe. More iteration space tiling. In *Supercomputing '88*, pages 655–664, Nov. 1989. 2.1.1

[195] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, The University of Maryland, 1995. 4.1.3, 1

[196] Y. Yamada, J. Gyllenhaal, G. Haab, and W. mei Hwu. Data relocation and prefetching for programs with large data sets. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 118–127. ACM SIGMICRO and IEEE Computer Society TC-MICRO, Nov. 30–Dec. 2, 1994. 2.4, 5.5

[197] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu. Compiler-assisted cache replacement: Problem formulation and performance evaluation. In *16th International Workshop on Languages and Compilers for Parallel Computing(LCPC'03)*, October 2003. 5.5

[198] Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of cache on the program execution. In *Information Visualization 2001*, pages 336–341, 2001. 1.2, 6.1, 7.1

[199] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of 6th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2002. 3.4

[200] Y. Zhong, S. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 79–90. IEEE, 2003. 3.4, 4.6.1, 4.7

[201] D. Zucker, R. Lee, and M. Flynn. An automated method for software controlled cache prefetching. In *System Sciences, 1998, Proceedings of the Thirty-First Hawaii International Conference on*, volume 7, pages 106–114, 1998. 5.5