A Unified Framework for Optimal Tile Size Selection

Lakshminarayanan Renganarayana
Colorado State University
ln@cs.colostate.edu

Abstract

Tiling is a widely used loop transformation for exposing/exploiting parallelism and data locality. Effective use of tiling requires selection and tuning of the tile sizes. This is usually achieved by developing cost models that characterize the performance of the tiled program as a function of tile sizes. All previous approaches to optimal tile size selection (TSS) are cost model specific. Due to this they are neither extensible (to richer program classes/newer architectures) nor scalable (to multiple levels of tiling).
Instead of proposing yet another TSS method, we take a fundamentally different approach and derive a unified TSS framework based on a mathematical property shared by almost all TSS models. Here are a few terms commonly used in the TSS models: total execution time, number of cache/TLB misses, size of communication messages/temporary storage, loop overhead, communication latency/bandwidth, cache size, miss penalty. Although these functions and parameters are very diverse, they all share a simple mathematical property, viz., *positivity*. We describe this property and use it to identify a class functions called posynomials which can be used to derive an efficient, scalable and cost-model independent framework for optimal TSS. Our framework not only unifies the variety of models proposed in the literature, but also lays the foundations to build more sophisticated models.

1 Problem and Motivation

The compute and data intensive kernels of several important applications are loops. Tiling [11,25,14,28] partitions the computations of the loop into groups called *tiles*. The tiles often have better data locality and serve as units of coarse grained parallelism. Typically multiple levels of tilings are used, e.g., a level of tiling to obtain coarse grained parallelism, a level for cache locality, and possibly another level for register reuse [5,29,21]. Recently, multi-level tiling has also been successfully used in automatic mapping of loops onto GPUs [3]. With the advent of multi- and many-core architectures multi-level tiling has become the standard design pattern for deriving high-performance implementations.
Effective use of tiling requires techniques for tile shape / size selection and tiled code generation. In this paper we focus on the key step of tile size selection (TSS) and our solution can be combined with the existing techniques for tiled code generation and tile shape selection. Tile size selection is an important problem due its performance impact: often there is a significant performance difference between a good tile size and a bad one. The importance is also evident from the extensive study of TSS methods across the last two decades.
Optimal Tile Size Selection is the problem of selecting the tile sizes that are *optimal* with respect to some cost model. For example, in the use of tiling to improve cache locality, consider the selection of sizes x and y which form the sides of a 2D tile. A widely used cost function is the number of cache misses. This cost function is used together with the constraint that the data accessed by a given tile-*tile footprint*-fits in the cache. The optimal TSS problem can be stated as follows:

$$\begin{aligned} &\text{select } x,y \text{ which minimize } Misses(x,y) \\ &\text{subject to } FootPrint(x,y) \quad < CacheCapacity \end{aligned} \tag{1}$$

where, $Misses(x,y)$ estimates the number of misses experienced with tile of size $x \times y$, $FootPrint(x,y)$ estimates the number of cache lines touched by a tile of size $x \times y$, and $CacheCapacity$ is the capacity of the cache in number of lines. The cost function together with the constraint is called the *cost model*. One can view the optimal TSS problem as a constrained optimization problem and in such a view the cost function is also referred to as the *objective function*. Solutions ranging from closed form solutions to heuristic algorithms and empirical search methods have been proposed (e.g., see [10,28,26,29]). However, all these solutions are cost model specific, i.e., their applicability is very sensitive to the particular properties of the functions used in the cost model. Due to this cost model specificity they are neither extensible to newer architectures / program class, nor scalable to multiple levels of tiling. The next section describes these limitations with an example.

2 Background and Related Work

In this section, we discuss the two primary limitations of current TSS methods, viz., (i) non-extensibility to newer architectures and program classes and (ii) non-scalability to multiple levels of tiling. We first describe the design process followed by current methods and using that highlight the limitations. All TSS solutions proposed currently in the literature follow a design process which can be summarized as follows:

1. *Design a cost model.* This includes the design of a cost metric (objective function) that estimates a desired quantity as a function of tile sizes and constraints that qualify tile sizes as valid or not. The cost models seek to estimate quantities that are related to the execution characteristics

and hence are inherently strongly tied to the class of programs and architectural features for which they are designed.

2. *Reason about the structure of the cost functions*. For example, one can check whether the objective function is linear or quadratic in terms of the tile size variables.
3. *Exploit the properties of functions to derive a closed form solution or a heuristic/search algorithm*.

As an illustration, consider the optimal TSS problem proposed by Andonov et al. [1]. They study the problem of tiling 2D iteration spaces with uniform dependencies for parallel SPMD style execution. After a detailed study of the class programs they want to tile, the architectural parameters, and the execution characteristics they come up with a cost model. The objective function $T(x,y)$ estimates the total (parallel) execution time of the tile program and the goal is to pick the tile sizes that minimize this metric. The objective function and the constraints can be abstractly viewed as

$$\text{min. } T(x,y) = \frac{A}{xy} + Bxy + Cx + \frac{D}{y} + E$$

$$\text{subject to } x,y \geq 1,\ x,y \in \mathbf{Z}$$

where x,y are the tile size variables and A,B,C,D,E are constants. Then they use the following reasoning to obtain a closed form solution: for $xy=K$, $K \in \mathbf{R}$ the function $T(x,y)$ monotonically decreases with x. As a result, the optimal solution is on certain boundaries of the feasible space, and using this information, one of the variables can be eliminated, yielding a closed form solution for x and y.

A subtle but important feature of the above process is the following: the cost model is strongly coupled to the program class/architectural features and the solution (method) is derived by exploiting the properties of the functions used in the cost model. Any extension of the cost model to a different architecture, richer program class, or to multiple levels of tiling, change the structure of the functions used in the cost model, and hence leave the solution (method) inapplicable. For example, an extension of the Andonov et al.'s model to a richer program class, viz., 3D iteration spaces, leaves their solution method inapplicable.

All optimal TSS solutions proposed in the literature are cost model specific and do not lend themselves to extensions. Any non-trivial extension typically requires an effort equal to or more than the earlier one, and are often publishable results (e.g., extension from direct mapped caches to set associative caches, from 2D to nD iteration spaces, etc.). Generally, one wants to use an TSS solution for a program class or architecture which is slightly different than the one considered by the author of the solution. But accounting for the differences lead to changes in the cost model, which leave the solution inapplicable. This is in fact an important reason for the popularity of exhaustive search (run the program for different tile sizes and pick the best).

The scalability limitation of the current approaches also stems from their strong dependence on the properties of the functions used in the cost model. For example, in a 2D one level tiling, the optimal TSS problem has the two tile sizes as variables and functions used in the cost models are of degree at most two (linear, quadratic, hyperbolic, etc.) and are easy to reason about. However, when we move to two levels of tiling there are four variables and functions of four variables with degree up to four are much harder to reason about. Single level empirical search solutions do not scale to multiple levels due to the combinatorial explosion of the search space size (induced by the increase in the number of tile size variables and also the possible values for the tile sizes). A simple solution is to find the tile sizes in a level-by-level approach (find tile sizes for one level, and use them to find the ones for the next level). However, with the multi-core architectures, there are complex interactions between different levels of resources (e.g., parallelism and locality) and due to this such a level-by-level approach is clearly sub-optimal. This sub-optimality of the level-by-level approach is also shown, in three different architectural scenarios, by Mitchell et al. [16].

To summarize, the cost-model specificity of the solution methods lead to their non-extensibility and non-scalability. Our framework overcomes these limitations by providing a cost-model independent solution method. When using our framework one does not have to perform the second and third steps of the traditional design process described above.

3 Uniqueness of the Approach

Instead of proposing yet another TSS method, we take a fundamentally different approach and seek to identify the common property shared by the variety of TSS models. Several authors have exploited particular properties such as linear, quadratic, hyperbolic, etc., of the cost functions to derive optimal TSS solutions. Instead of exploiting the specific properties of a cost model to derive a solution, we identify a fundamental mathematical property that is inherent to the TSS models and use it derive a unified TSS framework. Table 1 lists several functions and parameters that are used in TSS models. There is a fundamental mathematical property shared by all them, viz., *positivity*. All the machine and system parameters are positive quantities and the functions model positive quantities. The tile sizes which appear as variables in these functions are also positive. Essentially, the functions used in TSS models estimate positive quantities using positive parameters and positive variables. This positivity property might seem to be a simple one, but it has profound implications. This positivity property distinguishes the class of optimization problems that are solvable in polynomial time[1] and those that are not [4]. As we show in the coming sections we can use this property as a basis to identify a class of polynomials called *posynomials* which can be used to formulate optimal TSS problems that can be solved efficiently.

Machine and system parameters used in models	**Functions modeling quantities of interest**
Cache/TLB miss penalty Cache/TLB sizes Number of registers Number of functional units	Tile volume Number of tiles Number of cache misses Cache/register foot print

Latency of functional units	Idle time in parallel execution
Network latency	Communication volume
Network bandwidth	Loop overhead
MPI Call start-up cost	Temporary storage size
	Array pad size

Table 1: Parameters and functions that are widely used in TSS models.

Our framework provides a single unified solution for TSS which is useful in the model-based approach as well as the hybrid model-driven empirical search approach. In the later case, our framework would be used in the first filtering or starting point selection phase [30]. There by, our framework is a good candidate for inclusion in compilers as well as auto-tuners. Further, it is insightful to find that such a unified framework can be derived by exploiting a simple but fundamental property shared by all TSS models.

4 Results and Contributions

We would like to emphasize that the goal of this paper is not to show the benefits of loop tiling-several authors across two decades have shown that. Our goal is to propose an efficient framework which not only unifies the variety of models proposed in the literature, but also lays the foundations to build more sophisticated models. To this end, we substantiate our claim by (i) taking five different TSS models proposed in the literature by different authors and casting them into our framework and (ii) showing that almost all the cost functions used in the TSS literature are posynomials. This paper makes the following contributions:

- We identify the fundamental positivity property shared by many mathematical expression and terms used in a wide variety of TSS models. Based on this property, we identify a class of functions called posynomials which can serve as building block for modeling TSS problems.
- By formulating a class of non-linear optimization problems using posynomials, we propose an efficient, scalable and cost-model independent framework for optimal tile size selection.
- We show that almost all the tiling models proposed in the literature can be cast into our framework. We also show that several cost functions from the literature used for TSS are already posynomials. A detailed formal proof of the wide applicability of our framework is shown in [18] via a reduction of five different tiling models (from a wide range of tiling contexts) to our framework.
- We have implemented a MATLAB based tool for using posynomials to model and solve optimal TSS problems. We present experimental results on the running time of our tool on TSS models from the literature.

4.1 Posynomials and Geometric Programs

Monomials and posynomials are used in building the cost models. TSS models built with posynomials can be transformed into a particular class of numerical convex optimization problems called *Geometric Programs* (GP) which can solved efficiently [8,4]. We refer to GPs solved for integer solutions as Integer Geometric Programs (IGP).

4.1.1 Posynomials

Let x denote the vector $(x_1,x_2,\ldots,x_n)$ of n real, positive variables. A function f is called a *posynomial* function of x if it has the form

$$f(x_1,x_2,\ldots,x_n)=\sum_{k=1}^{t} c_k x_1^{a_{1k}} x_2^{a_{2k}} \ldots x_n^{a_{nk}}$$

where $c_j \geq 0$ and $a_{ij} \in \mathbf{R}$. Note that the coefficients c_k must be non negative, but the exponents a_{ij} can be any real numbers, including negative or fractional. When there is exactly one nonzero term in the sum, i.e., $t=1$ and $c_1 > 0$, we call f a *monomial* function.[2] For example, $0.7+2x_1/x_3^2+x_2^{0.3}$ is a posynomial (but not a monomial); $2.3(x_1/x_2)^{1.5}$ is a monomial (and, hence a posynomial); while $2x_1/x_3^2-x_2^{0.3}$ is neither.

Monomials and posynomials enjoy a rich set of closure properties, which are very useful in composition of smaller (say single level) TSS models to build larger (multi-level) ones. Monomials are closed under product, division, non-negative scaling, power and inverse. Posynomials are closed under sum, product, non-negative scaling, division by monomials, and positive integer powers.

4.1.2 Efficient solutions via Convex Optimization

Recent advances [4] in convex optimization provide efficient polynomial time solution methods. GPs can be transformed into convex optimization problems using a variable substitution and solved efficiently using polynomial time interior point methods [12,4]. The positivity property of the posynomials is extensively exploited in this transformation of GPs to convex optimization problems. The computational complexity of solving GPs are similar to that of solving linear programs. Continuous real solutions can be found very quickly in polynomial time. Integer solutions need a branch and bound style algorithm, which in the worst-cast can take exponential time. However, we have found (*cf.* Sec 4.3) that for optimal TSS problems the IGPs are very small (few tile size variables and constraints) and solutions can be found quickly. Further, in the context of TSS, it is very common to solve for real solutions and round them to obtain integer solutions. In such an approach we can obtain the solution in polynomial time irrespective of the complexity of the model.

4.2 Posynomials and TSS models

Posynomials are well suited for modeling TSS problems. The appropriateness is evident from the fact that almost all TSS cost functions considered in the literature turn out to be posynomials. A few of them are discussed in this section.

- **Models for data locality.** In general, as observed by Hsu and Kremer [10], the objective functions used in the context of TSS are all functions of the tile variables, cache capacity and cache line size. Due to the positivity of both the tile size variables and the cache parameters, these functions turn out to be posynomials. For example, as shown in Table 2 the cost functions used in several widely used TSS models [13,7,9,17,6,27,22,16] turn out to be posynomials. In addition to this, the TSS models used in the IBM XL compiler as described in [23] and the multi-level data locality tiling model proposed in [19] use posynomials and can be reduced to an IGP.
- **Models for parallelism.** Similar to data locality models, several important and popular models used in TSS for parallelism can be reduced to IGPs. Here, the TSS models are formulated with quantities such as tile volume, number of tiles, idle time in parallel execution, etc. and parameters such as network bandwidth/latency, MPI communication call cost, etc. Due to the positivity of the parameters and quantities they use and the tile size variables, these functions turn out to be posynomials. In particular, the commonly used communication minimal tiling of uniform dependence programs with rectangular tiles [28] can be cast as an IGP, since the communication volume can be expressed as a non-negative combination of the tile facets. Other models that can also be reduced to IGPs include optimal orthogonal tiling [1], 2D semi-oblique tiling [2] and the Multi-level tiling model for 3D stencil computations [21].
- **Register tiling, auto-tuners, and Multi-level cost models.** The register tiling models proposed in [24] and [20] can be reduced to IGPs. The cost model used for generating high performance BLAS as described in [29] and the multi-level cost model [16] used for quantifying the multi-level interactions of tiling, can also be directly reduced to IGPs.

The fact that such a large number of TSS models-proposed across two decades by a several different authors-can all be reduced to the posynomial based TSS framework shows its generality and wide applicability. We have given in [18] detailed evidence of the wide applicability of our framework via a reduction of five different tiling models (from a wide range of tiling contexts) to our framework. Due to space limitations we do not discuss these reductions here.

Cost Model Reference	Cost function used for selecting the optimal tile size
ESS [9]	$C/(h*w)$
LRW [13]	$1/h+1/w+(2h+w)/C$
TSS [7]	$(2h+w)/h*w$
EUC [22]	$1/h+1/w$
MOON [17]	$1/h+1/w+(h+w)/C$
TLI [6]	$1/h+1/w+(h+w)/C+h*w/C^2$
WMC [27]	$C/h*w$
MHCF [16]	$(1/h+1/w)(1/n+1/l)+2/(h*w)$

Table 2: Cost functions used in the literature for optimal cache locality tiling are shown, where C is the cache size, h,w represent the height and width of the rectangular tile, n represents the size of a 2D array and l represents the cache line size. A simple inspection shows that they are all posynomials. This table is derived from Hsu and Kremer [10,table 2].

4.3 PosyOpt Framework

We have implemented the TSS framework as a tool called PosyOpt. The implementation uses MATLAB and YALMIP [15] a tool which provides a symbolic interface to several optimization tools on top of MATLAB. The symbolic interface allows a high level specification of the optimal TSS problems. The overall structure of our tool PosyOpt is shown in Figure 1. The optimal TSS problems are specified at a high level using posynomials as a IGP. These problems are then automatically transformed to a convex optimization problem. The transformed problem is then fed to the convex optimization solver of MATLAB and solved for real solutions. Integer solutions are found via a branch-bound algorithm which internally uses the MATLAB solver for solving continuous relaxations. The output of our tool is the set of optimal tile sizes.

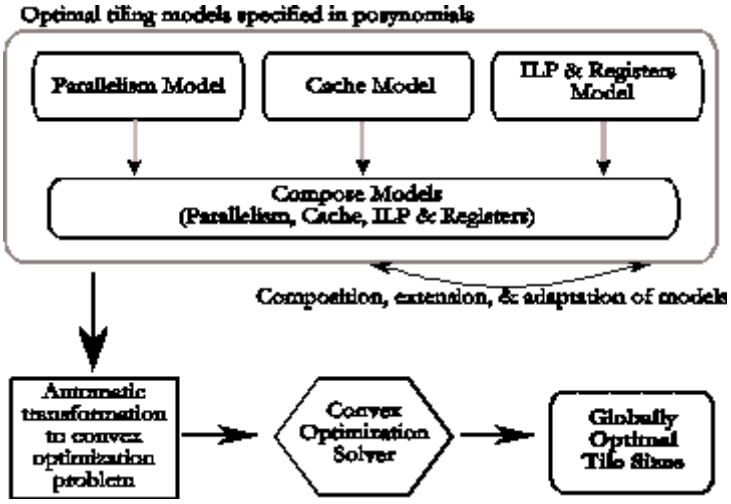

Figure 1: Overall structure of the PosyOpt tool.

Note that the specification and subsequent refinement and extensions are performed at the posynomial level (*cf.* top box in Figure 1). These steps are done without any concern about the solution method. The only concern is that the specifications and extensions use posynomials. Further, as shown in the Figure 1 (top box) different models can be combined or composed together to form multi-level tiling models. We have found the closure properties of monomials and posynomials to be very useful during the extensions and compositions.

4.3.1 Running time experiments

Solution Type	Real	Real	Integer	Integer
	Yalmip	Solver	Yalmip	Solver
tm1n2	0.087	0.045	0.09	0.06
tm1n3	0.1	0.05	0.11	1.21
tm2n2	0.09	0.07	0.09	0.26
tm2n3	0.1	0.05	0.11	0.05
SOB	0.08	0.04	0.08	0.04

Table 3: Running times in seconds of the PosyOpt tool on various optimal TSS problems.

Using our tool, we formulated and solved a variety of single level and two level optimal TSS problems. The number of variables in any optimization problem is determined by the loop nest depth and the number of levels of tiling. For example, a 2D loop nest tiled twice, would have 4 variables in the optimal TSS problem. The time our tool takes to find the integer and real (continuous) solutions are shown in Table 3. The time taken to find real and integer solutions are shown with the break up of the time taken by Yalmip to process the problem specified in symbolic form and the time taken by the solver to find the solutions. For the first four problems (rows) `tmXnY` means a loop nest of depth Y tiled X number of levels and are taken from data locality tiling models in [19]. For example, tm2n3 means a 3D loop nest tiled twice. The last row is from a TSS problem for single level tiling of 3D stencil computations for parallelism, taken from [21]. We can note that on the average the overhead for Yalmip's preprocessing is around 0.1 seconds. The average solver time to find the real solutions is around 0.05 and the solver time for integer solutions is between 1.21 and 0.06 seconds. One can observe that for many of the cases the solver time for real and integer solutions are very close. These are the cases for which a rounding of the real solution is equal to the integer solution. For the uses in the context of an auto-tuner or a compiler, the current speed of our tool seems very reasonable, particularly given the ease with which the problem can stated, refined, and solved.

4.4 Discussion

We got the insight about the positivity property of the TSS models after we developed three different cost model specific TSS methods [19,20,21]. While deriving these solutions we observed that we were repeatedly using the same class of functions (posynomials) and the same optimization problem (GPs). This led us to seek for the common properties shared by the TSS models.
We have proposed a framework based on a simple yet fundamental property of functions used in TSS models. Our framework not only generalizes the TSS models proposed in the literature, but also provides the foundation for developing more sophisticated and particularly multi-level tiling models. The closure properties of posynomials can be directly exploited to build multi-level tiling models by composing the well understood single level models. We are currently developing one such posynomial based multi-level TSS models for OpenMP based parallel programs. Our framework provides a single unified solution for TSS that is useful in both compilers and auto-tuners.

References

[1]

R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *Journal of Parallel and Distributed Computing*, 45(2):159-165, September 1997.

[2]

Rumen Andonov, Stephan Balev, Sanjay V. Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(9):944-960, 2003.

[3]

Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1-10, New York, NY, USA, 2008. ACM.

[4]

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press. (Online version available at: `http://www.stanford.edu/~boyd/cvxbook.html`), 2004.

[5]

Larry Carter, Jeanne Ferrante, and Susan Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 239-245, Washington, DC, USA, 1995. IEEE Computer Society.

[6]

Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th international conference on Supercomputing*, pages 492-499. ACM Press, 1999.

[7]

Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279-290. ACM Press, 1995.

[8]

R.J. Duffin, E.L. Peterson, and C. Zener. *Geometric Programming - Theory and Applications*. John Wiley, 1967.

[9]

Karim Esseghir. Improving data locality for caches. Master's thesis, Rice University, September 1993.

[10]

Chung hsing Hsu and Ulrich Kremer. A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279-294, 2004.

[11]

F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319-328. ACM, Jan 1988.

[12]

K. O. Kortanek, Xiaojie Xu, and Yinyu Ye. An infeasible interior-point algorithm for solving primal and dual geometric programs. *Math. Program.*, 76(1):155-181, 1997.

[13]

Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63-74. ACM Press, 1991.

[14]

Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm (with retrospective). In *Best of PLDI*, pages 442-459, 1991.

[15]

J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. Available from http://control.ee.ethz.ch/~joloef/yalmip.php.

[16]

N. Mitchell, N. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641-670, 1998.

[17]

S. Moon and R. Saavedra. Hyperblocking: A data reorganization method to eliminate cache conflicts in tiled loop nests. Technical Report TR-98-671, University of Southern California, February 1998.

[18]

Lakshminarayanan Renganarayana. *Scalable and Efficient Tools for Multi-level Tiling*. PhD thesis, Department of Computer Science, Colorado State University, 2008.

[19]

Lakshminarayanan Renganarayana and Sanjay Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 18, Washington, DC, USA, 2004. IEEE Computer Society.

[20]

Lakshminarayanan Renganarayana, Ramakrishna Upadrasta, and Sanjay Rajopadhye. Optimal ILP and register tiling: Analytical model and optimization framework. In *LCPC 2005: 12th International Workshop on Languages and Compilers for Parallel Computing*. Springer Verlag, 2005.

[21]

Lakshminarayanan Renganarayanan, Manjukumar Harthi-kote, Rinku Dewri, and Sanjay Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[22]

Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction*, pages 168-182. Springer-Verlag, 1999.

[23]

V. Sarkar. Automatic selection of high-order transformations in the ibm xl fortran compilers. *IBM J. Res. Dev.*, 41(3):233-264, 1997.

[24]

Vivek Sarkar. Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 29(5):545-581, 2001.

[25]

R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, NASA Ames Research Center, Aug 1990.

[26]

R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1-27. IEEE Computer Society, 1998.

[27]

M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *29th International Symposium on Microarchitecture*, December 1996.

[28]

Jingling Xue. *Loop Tiling For Parallelism*. Kluwer Academic Publishers, 2000.

[29]

K. Yotov, Xiaoming Li, Gang Ren, M. J. S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93:358-386, 2005.

[30]

Kamen Yotov, Keshav Pingali, and Paul Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141-150, New York, NY, USA, 2005. ACM.

Footnotes:

[1]Use of polynomial functions with this property leads to convex optimization problems which can be solved for real solutions in polynomial time. On the other hand, optimization problems formulated with arbirtrary polynomials are not solvable in polynomial time.

[2]Note that this definition of monomial is different from the standard one used in algebra.