Static Analysis of Parameterized Loop Nests for Energy Efficient Use of Data Caches

Paolo D'Alberto, Alexandru Nicolau, Alexander Veidenbaum, and Rajesh Gupta *

Information and Computer Science
Center for Embedded Computer Systems
University of California at Irvine
{paolo,nicolau,alexv,rgupta}@ics.uci.edu

Abstract. Caches are an important part of architectural and compiler low-power strategies by reducing memory accesses and energy per access. In this paper, we examine efficient utilization of data caches for low power in an adaptive memory hierarchy. We focus on the optimization of data reuse through the static analysis of line size adaptivity. We present an approach that enables the quantification of data misses w.r.t. cache line size at compile-time. This analysis is implemented in a software package STAMINA. Experimental results demonstrate effectiveness and accuracy of the analytical results compared to alternative simulation based methods.

1 Introduction

In modern uniprocessor systems the memory hierarchy is an important concern of performance, area and energy. It is also the component requiring most of the die area in systems-on-chip and it is the principal power consumer, accounting for as much as 20-50% of the total chip power [12, 8]. In recent years, there has been a great effort on the engineering of several levels of cache, to reduce the impact on performance/power of caches. The focus of our work on memory hierarchy is adaptivity in cache subsystems. We have built an architecture that enables static and dynamic adaptation of memory hierarchy: its configuration and policies [18]. In this paper, we focus on (compiler-driven) data cache line size adaptation [17, 1, 18]. In fact, the architecture is able to change dynamically the line size (by hardware monitoring or application instruction) during the execution of the application. To exploit fully the potential of this adaptation, we need a way to target it, that is, (statically) determine the application cache behavior to trace adaptation for maximum performance and/or minimum energy dissipation.

Related work on cache behavior analysis can be distinguished in *profiling-based* and *static* approaches. Profiling has been used to determine the memory behavior by direct measure. Varying some parameters of the architecture, the direct measure quantifies the variation of the memory performance [11]. Static cache analyzers are independent from the *inputs* and focus on the analysis of perfect loop nests [7, 19]. In [7], the authors propose to model the cache misses of memory references by equations, *Cache Miss Equations*. Then every iteration (or a sampled version) in the loop nest is checked whether it satisfies the equations or it does not. The approaches count the solutions of the equations to achieve an estimation of the number of cache misses.

There are two limitations in the current static approaches: 1) The loop nest bounds must be known at compile time. This is not realistic because they are often parameterized and it is not practical, because they can be very large. 2) The analyzable loops are *sensitive* to tiling loop transformation. For example, if tiling is performed on the three-loop-algorithm for matrix multiplication and the tile sizes do not divide evenly the loop bounds, the inner loops bounds cannot be represented by affine functions. The resulting nest is not analyzable.

To overcome these limitations, in this paper we propose a static approach to investigate perfect loop nests and determine the relation between line size and number of misses on a per-nest-base. The analysis result is annotated in the code and it can be used at run time to set the line size.

The paper is organized as follows. In Section 2 we present two models for energy dissipation and execution time in function of the miss ratio. In Section 3 we introduce notations about loop nests and cache equations. In Section 4 we introduce the theoretical frame work and our approach. Finally, in Section 5 we show the results of our analysis for three representative examples.

2 Energy and Line Size

In this section we explore the line size effect on energy dissipation, using a theoretical model based on the cache models presented and used in [12, 16, 9, 15]. The energy dissipation per access on a cache, C, with line size, L, is $E_{acc}(L) = E_{tag} + E_{line} + E_{status}$ [12, 16]. E_{tag} is the energy to determine row and line in the cache (for adaptive fetch size, it is constant). E_{line} is the energy to precharge, buffer and deliver on the bus the line accessed. E_{status} is the energy to check the cache line status (dirty,

* Supported by AMRM DABT63-98-C-0045

present). In direct mapped caches, the dominant term is $E_{line} = R_i \lceil L/L_{i_{bus}} \rceil$ where R_i is a constant for the i-th level of the memory hierarchy and $L_{i_{bus}}$ is the physical bus width [9]. The energy for a hit on L1 is independent from the line size, because of subbanking [8,16]. If $\lambda(L)$ is the data miss ratio in function of the line size and $|M_{acc}|$ the memory accesses, the energy dissipated for a two level memory hierarchy is:

$$E(L) = |M_{acc}|[(1-\lambda(L))R_1 + \lambda(L)R_2\lceil \frac{L}{L_{2_{bus}}} \rceil] \quad (1)$$

The access time formula is very similar to the energy dissipation formula. The access time on $L1$ is constant [8] but for other levels it depends on the line size. The access time is:

$$T(L) < |M_{acc}|[(1-\lambda(L))T_1 + \lambda(L)T_2\lceil \frac{L}{L_{2_{bus}}} \rceil] \quad (2)$$

This is an upper bound since some of the accesses may be pipelined and the access time can be hidden.

For example, with $\lambda(L) = 1/L$ and $L > L_{2_{bus}}$ the minimum is when $L = -\frac{K_1 L_{2_{bus}}}{K_2}$ with K_i equal to either T_i or R_i. This implies that in general the optimal line size to achieve minimum access time is different from the optimal line size to achieve minimum energy dissipation (for the memory hierarchy system). A very interesting case is when the line size is shorter than the bus width ($\lceil \frac{L}{L_{i_{bus}}} \rceil = 1$), because the line minimizing the miss ratio is *optimal* for both performance and energy.

In the rest of the paper we focus our analysis on line size optimization. This optimization is a classical trade-off between reuse and conflicts. We report on the analysis line model that enables the trade-off analysis, therefore quantifying the relation between line size and data miss ratio ($\lambda(L)$).

3 Background

A perfect loop nest of depth k [14] determines a set (*iteration space*) of integral points (*iteration points*), in $\mathbb{N}^k$. The loop order specifies a strong order between any two iteration points $\bar{\mathbf{j}} = (j_0, \ldots, j_{k-1})$ and $\bar{\mathbf{i}} = (i_0, \ldots, i_{k-1})$. $\bar{\mathbf{i}}$ *precedes* $\bar{\mathbf{j}}$, and we indicate as $\bar{\mathbf{i}} \lhd \bar{\mathbf{j}}$, if it exists a $0 \leq l \leq k-1$ so that $i_n = j_n$ for every $n < l$ and $i_l < k_l$. The inequality for two points $\bar{\mathbf{i}} \unlhd \bar{\mathbf{j}}$ is verified if either they coincide or $\bar{\mathbf{i}} \lhd \bar{\mathbf{j}}$. A geometrical order can be inferred too. $\bar{\mathbf{i}}$ is smaller than $\bar{\mathbf{j}}$, $\bar{\mathbf{i}} < \bar{\mathbf{j}}$, if $i_n \leq j_n$ for every n but a l so that $i_l < j_l$. Graphically, a point $\bar{\mathbf{i}}$ in the iteration space determines a unique bounded polytope ($\{\bar{\mathbf{j}} | \bar{\mathbf{j}} \leq \bar{\mathbf{i}}\}$), a point is smaller than another if a bounded polytope is contained in the other. It is easy to see that if $\bar{\mathbf{i}} < \bar{\mathbf{j}}$ then $\bar{\mathbf{i}} \lhd \bar{\mathbf{j}}$, but not vice versa. Informally, the first coordinate i_0 of an iteration point $\bar{\mathbf{i}}$ is associated with the outer loop of the nest and the last coordinate i_{k-1} is associated with the inner loop.

Formally, *the iteration space* is a bounded polytope (lattice). The iteration space is the polytope $\{\bar{\mathbf{i}} | \bar{\mathbf{0}} \unlhd \bar{\mathbf{i}} \unlhd N(\bar{\mathbf{n}})\}$, in short $P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})}$, where $N(\bar{\mathbf{n}}) = F\bar{\mathbf{n}} + G\bar{\mathbf{p}}$ with F and G matrices of size $k \times k$ and $\bar{\mathbf{p}}$ parameters vector.

Given a point $\bar{\mathbf{t}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})}$ and a vector $\bar{\mathbf{r}}$, it is common to investigate the bounded polytope $P^r(\bar{\mathbf{t}}) \equiv \{\bar{\mathbf{j}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})} | \bar{\mathbf{t}} - \bar{\mathbf{r}} \lhd \bar{\mathbf{j}} \unlhd \bar{\mathbf{t}}\}$. It is an interval in the iteration space. Indeed, $P^r(\bar{\mathbf{t}}) = P_{\bar{\mathbf{i}} \unlhd \bar{\mathbf{t}}} - P_{\bar{\mathbf{i}} \lhd \bar{\mathbf{t}} - \bar{\mathbf{r}}}$ is the difference of two parameterized polytopes containing the origin. As introduced in [21], a reference R_B of a k-dimensional array B, i.e. $B[i_0] \ldots [i_{k-1}]$, has temporal reuse if in different iteration points the same memory location is accessed: $Addr(R_B(\bar{\mathbf{i}})) = Addr(R_B(\bar{\mathbf{j}}))$. A vector $\bar{\mathbf{r}}$ summarizes the reuse such as for every iteration point $\bar{\mathbf{i}}$, $Addr(R_B(\bar{\mathbf{i}})) = Addr(R_B(\bar{\mathbf{i}} + \bar{\mathbf{r}}))$. We assume that the address of a reference is a linear function: $Addr(R_B(\bar{\mathbf{i}})) = \mathbf{B}\bar{\mathbf{i}} + \mathbf{B}_{-1}$. For every reuse vector, $\mathbf{B}\bar{\mathbf{r}}$ is 0. Spatial reuse is attained when elements of the same line is accessed. Temporal reuse is a particular case of spatial reuse.

If reference R_B interferes with reference R_A, the reuse of R_A is *prevented* by R_B. The interference between memory references is investigated by the *Cache Miss Equation* (CME) model (see [7]).

Definition 1. *Given two array references R_A (*interferer*) and R_B (*interferee*) of the arrays A and B respectively, we define a subset of points EQ_1.*

$$EQ_1 \equiv \begin{cases} \mathbf{A}\bar{\mathbf{i}} + \mathbf{A}_{-1} = \mathbf{B}\bar{\mathbf{t}} + \mathbf{B}_{-1} + n\mathcal{C} + l + D\bar{\mathbf{p}} \\ \text{with } \bar{\mathbf{i}} \in P^r(\bar{\mathbf{t}}) \text{ and } \bar{\mathbf{t}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})} \end{cases} \quad (3)$$

$\mathbf{A}$ and $\mathbf{B}$ are integer matrices of size $1 \times k$ describing the index computation. $\mathcal{C}$ is the cache size in bytes, $|l| < L-1$ is the offset in the cache line and L is the cache line size. The free variable $n \neq 0$ describes the distance in number of cache size blocks between the references. $\bar{\mathbf{p}}$ is a vector of parameters. $\bar{\mathbf{r}}$ is a reuse vector for R_B.

In a direct mapped cache, if the equation has solution, there is a miss. A k-way cache needs k interferences before to have a miss. To estimate the number of misses we need to count the interferences at least L bytes apart. We analyze references with very short reuse vectors [13]. The interferer has a few chances to interfere with different memory locations. Therefore, the problem is very often reduced into the existence of interference.

4 The Parameterized Loop Analysis

Let us focus on the determination of interferences and, indirectly, on a subset of *cold misses*.[1]. When $\bar{\mathbf{p}}$ is con-

[1] We do not consider *capacity misses* because they should be independent from the line size.

sidered constant, Equation 3 can be rewritten in.

$$EQ_2 \equiv \begin{cases} A_{-1} - B_{-1} + \sum_{k=0}^{d-1}(A_k i_k - B_k t_k) = n\mathcal{C} + l \\ \text{with } \bar{\mathbf{i}} \in P^r(\bar{\mathbf{t}}) \text{ and } \bar{\mathbf{t}} \in P_{\bar{\mathbf{i}} \le N(\bar{\mathbf{n}})} \end{cases} \quad (4)$$

We define the *interference density*, $\rho_E \in [0,1]$, of an equation as the ratio of iteration points where the interference equation is satisfied over all the iteration points.

When the reuse vector is short (distance equal to 1) -and this is often the case for code optimized to exploit spatial locality- the Equation 4 can be simplified:

$$AB_{-1} + \sum_{k=0}^{d-1}(AB_k t_k) = n\mathcal{C} + l \text{ with } AB_k = A_k - B_k \quad (5)$$

Property 1. If $AB_{-1} + \sum_{k=0}^{d-1}(AB_k t_k) = n\mathcal{C} + l$ has solution and $AB_k \textbf{ mod } \mathcal{C} = 0$ with $0 \le k \le d-1$, then $\rho_E = 1$.

Proof. $\lfloor \frac{AB_{-1} + \sum_{k=0}^{d-1}(AB_k t_k)}{\mathcal{C}} \rfloor = \lfloor \frac{AB_1}{\mathcal{C}} + \frac{\sum_{k=0}^{d-1}(AB_k t_k)}{\mathcal{C}} \rfloor = \lfloor \frac{AB_1}{\mathcal{C}} + \sum_{k=-1}^{d-1}(\frac{AB_k}{\mathcal{C}} t_k) \rfloor = \lfloor \frac{AB_1}{\mathcal{C}} \rfloor + \sum_{k=-1}^{d-1} n_k t_k$. If Equation 5 has solution, it is independent from $\bar{\mathbf{t}}$, every iteration point is solution.

By Property 1 we need to focus on the following equation.

$$EQ_3 \equiv \begin{cases} AB_{-1}^m + \sum_{k=0}^{d-1}(AB_k^m t_k) = n\mathcal{C} + l \\ \text{with } AB_k^m = AB_k \textbf{ mod } \mathcal{C} \end{cases} \quad (6)$$

We distinguish integer solutions and rational solutions. The existence of integer solution assures the existence of rational solutions. We show a rational solution space such that contains every integer solution. We then determine the density in the rational space.

Given $\dot{\mathbf{q}}$ the smallest rational solution to Equation 6 a *grid* consists of the solution points $\mathcal{G}(\overline{\mathbf{q}}) = \{\overline{\mathbf{g}} |$ for any k $g_k = \dot{q}_k + \frac{\mathcal{C}}{AB_k^m} p_k$with p_k natural number$\}$. A *grid cell* is the smallest polygon that has all vertices in the grid. Given an integer l, a *band* is the set of rational points $\mathcal{B}(l) = \{\Delta\overline{\mathbf{b}} | -L < l + \sum_{k=0}^{d-1} AB_k^m \Delta b_k < L\}$. Note that the origin always belongs to a band when $-L < l < L$. A *band cell* is the polygon bounded by the two hyper-planes $-L = l + \sum_{k=0}^{d-1} AB_k^m \Delta b_k$ and $L = l + \sum_{k=0}^{d-1} AB_k^m \Delta b_k$ and the planes $\forall k \ne j, \Delta b_k = -L$ and $\Delta b_k = L$ for any $0 \le j \le d-1$. For every grid point we can determine a band ($\mathcal{B}(l + AB_{-1}^m)$). Every point in the band has the same solution value for n. In the band, we can distinguish different band cells. The space determined by the grid and the bands on the grid points is dense as formalized in the following property.

Property 2. For any integer solution $\overline{\mathbf{z}}$, there is a grid point in the band passing through $\overline{\mathbf{z}}$.

Proof. By definition, $AB_{-1}^m + \sum_{k=0}^{d-1} AB_k^m z_k = n_z\mathcal{C} + l_z$; we can represent $z_k = p_{z_k}\frac{\mathcal{C}}{AB_k^m} + \gamma z_k$ where $\gamma z_k = z_k \textbf{ mod } \frac{\mathcal{C}}{AB_k^m}$. Therefore, $AB_{-1}^m + \sum_{k=0}^{d-1} AB_k^m (p_{z_k}\frac{\mathcal{C}}{AB_k^m} + \gamma z_k) = AB_{-1}^m + \sum_{k=0}^{d-1} p_{z_k} + \sum_{k=0}^{d-1} AB_k^m \gamma z_k = n_z\mathcal{C} + l_z$. We have that

$$\begin{cases} n_z = \sum_{k=0}^{d-1} p_{z_k} + \lfloor \frac{AB_{-1}^m + \sum_{k=0}^{d-1} AB_k^m \gamma z_k}{\mathcal{C}} \rfloor \\ l_z = (AB_{-1}^m + \sum_{k=0}^{d-1} AB_k^m \gamma z_k) \textbf{ mod } \mathcal{C} \end{cases} \quad (7)$$

We can see that $-(d-1) \le \lfloor \frac{AB_{-1}^m + \sum_{k=0}^{d-1} AB_k^m \gamma z_k}{\mathcal{C}} \rfloor \le d-1$ because for every k we have $AB_k^m \gamma z_k = z_k \textbf{ mod } \mathcal{C}$. There are several points in the neighborhood of $\overline{\mathbf{z}}$ and in the grid that have the same solution in n, therefore in the band passing through $\overline{\mathbf{z}}$.

For any grid cell there is only one band splitting the cell in two, so that two vertices are apart. The band is determined by two planes, and by construction they have same inclination of the plane passing through the the grid points. See Figure 1 for an example in a 2-dimensional space. Now we are ready to determine the

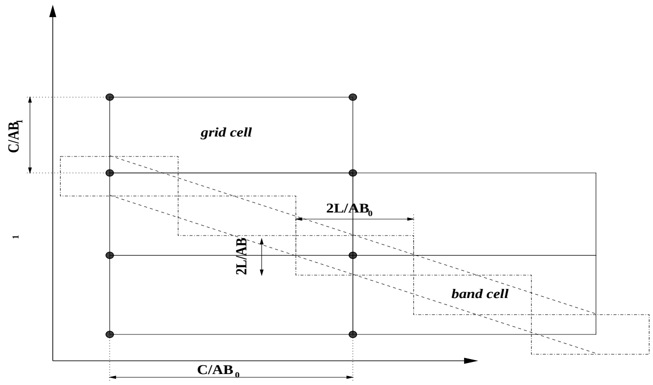

Fig. 1. Grid cells and band cells in a plane. In a 2-dimensional space the grid cell is a rectangle and the band is a between two lines crossing the grid cell on only two grid points. Two different bands are crossing the remaining two vertices.

solution density.

Property 3. Every grid cell has $\frac{\mathcal{C}^d}{\prod_{k=0}^{d-1} AB_k^m}$ solution points.

Property 4. Every band cell has at most $\frac{(2L)^d}{\prod_{k=0}^{d-1} AB_k^m}$ solution points.

Property 5. Every grid cell intersects at most three bands and at most $\frac{1}{2^{d-1}}(\frac{\mathcal{C}}{2L})^{d-1}$ band cells.

Proof. When $d = 2$, the solution space is a line. The line intersecting the cube (rectangle) is a diagonal. The number of possible integer solutions are not more than the number of integer coordinates in each dimension. Therefore, they are $\min_{i\in\{0,1\}}(\frac{C}{AB_i}\frac{AB_i}{2L})$. When $d > 2$, the solution space is a (d-1)-dimensional plane. It intersects d vertices of the grid cell. We project the grid cell and the solution on any plane $t_i = 0$. If the projections on the plane $t_j = 0$ is projection with minimum integer solutions, the number of integer solutions must be at most 1/2 of the minimum integer solutions of the projections multiplied by $\lfloor \frac{C}{2L} \rfloor + 1$. Therefore, the upper bound of integer solution is $f(d) = \frac{1}{2}(\frac{C}{2L})f(d-1)$ with $f(2) = \frac{C}{2L}$. $f(d)$ is $\frac{1}{2^{d-1}}\prod_{i=0}^{d-2}(\frac{C}{2L})$.

Theorem 1. *If $AB_{-1} + \sum_{k=0}^{d-1}(AB_k t_k) = n\mathcal{C} + l$ has solution and AB_k **mod** $\mathcal{C} \neq 0$ with $0 \le k \le d-1$ and $C \ge 4L$, then $\rho_E \le \frac{1}{2^{d-1}}\frac{2L}{C}$.*

ρ_E is the ratio of the number of solutions in a band intersecting a grid cell and the points in a grid cell.

When the reuse vector has distance h, it is possible to write the Equation 4 as a system of h equations. Each equation differs for a constant term. We approximate the density as $\rho = \min(1, h\rho_E)$.

The *interference existence* for an equation is a function $\chi_{P(L)}$ where $P(L)$ is a polyhedron determined by the interference equation and for which the line size L is parameter. If $P(L)$ has an integral solution, $\chi_{P(L)} = 1$; if it has not integral solution $\chi_{P(L)} = 0$. χ is a monotone increasing function. Indeed, if $L_0 \le L_1$, then $\chi_{P(L_0)} \le \chi_{P(L_1)}$. If there is interference for a line size L_i, there is interference for any larger line size.

Corollary 1. *If $m > 0$ is the number of coefficients in Equation 3 so that $(A_i - B_i)$ **mod** $C \neq 0$, then the cache miss ratio is at most $\rho = \chi_{P(L)}\min(1, \frac{h}{2^{m-1}}\frac{2L}{C})$. If $m = 0$, the cache miss ratio is at most $\chi_{P(L)}$.*

4.1 Reduction to Single Reference Interference

We now show how the general case can be reduced to the simplest case. The simplest case is as follows: there is an iteration space with $|I|$ iteration points; there is a reference with only one interference equation EQ_1 (one reuse vector of size h and one interferer); the interference polyhedron is function of the line size and it is denoted as $P(L_i)$. The number of misses is $|I| * \rho * \chi_{P(L)}$.

1. A reference R_A has k interferers, and R_A has just one reuse vector $\bar{\mathbf{r}}$. Every interference equation has density solution, ρ_i, and the solution existence function, $\chi_{P_i(L)}$ $(0 \le i \le k)$. Since the interferences due to different interferers are independent to each other, we can add their contribution: $\mu(L) = \sum_{i=0}^{k}\rho_i\chi_{P_i(L)}$. [2] We identify the function $\mu(L)$ as *interference density per reference*. The upper bound to the number of misses for a direct mapped cache is $|I| * \mu(L)$. If we would model a m-way associative cache, we could consider as estimation of the number of misses $|I|\lfloor\frac{\mu(L)}{m}\rfloor$ (this is an approximation, not an upper bound unless $\forall i, \rho_i = 1$).
2. R_A (interferee) and R_B (interferer). R_A has multiple reuse $\{\bar{\mathbf{r}}_i\}_{0,m-1}$ so that $\bar{\mathbf{r}}_0 > \bar{\mathbf{r}}_1 > \cdots > \bar{\mathbf{r}}_{m-1}$. Therefore we have $P^{r_i}(\bar{\mathbf{t}}) \supset P^{r_j}(\bar{\mathbf{t}})$ with $i < j$, and $\cap_{i=0}^{k}P^{r_i}(\bar{\mathbf{t}}) = P^k(\bar{\mathbf{t}})$. We consider only the shortest reuse, because if the reuse is prevented, there is a miss. If it is not, there is no miss.
3. R_A has k interferers and m reuse vectors. A set of equations $E_0 \dots E_{k-1}$ represent the interferences with different references. For each equation we consider only the shortest reuse vector.

[2] Interferes are truly independent to each other if they are at least one cache line apart

For every reference we are able to quantify the interference. In the following, we investigate the effect of interference on spatial reuse. Larger line size increases interference (it decreases performance) but also spatial reuse.

4.2 Interference and Reuse Trade-off

In this section we consider the effect of the line size on cache reuse and the trade-off with conflicts. Every reference reuse vector has a type of reuse, i.e. *spatial* or *temporal*. If a reference has spatial reuse and no interference, the reference has a miss every $\frac{1}{\ell}$ access, where ℓ is the line size in data elements.[3] If Interference is present some of the reuse can be prevented. The *the miss density for spatial reuse* is $\eta(L) = \frac{1}{\ell} + \mu(L)$ if $\mu(L) < 1$, $\eta(L) = \mu(L)$ otherwise. It is always possible to label the references so that R_i with $0 \le i \le n-1$ are references with spatial reuse and R_i with $n \le i \le m-1$ with temporal reuse. The *density of the misses for the loop nest* is $\epsilon(L) = \sum_{i=0}^{n-1}\eta_i(L) + \sum_{i=n}^{m-1}\mu_i(L)$.

$\lambda(L) \sim |I| * \epsilon(L)$ is the number of misses for which the line size has any effect. It is an estimation of the effect of line size on cache performance.

5 STAMINA Implementation Results

The reuse and interference analysis is implemented in the software package StaMInA (*Sta*tic *M*odeling of *In*terference *A*nd reuse as a part of AMRM compiler suite). It is built on top of SUIF 1.3 compiler adapting the code developed in [7] and using *polylib* [20, 4, 3, 5]. We consider three examples to explore three important aspects of our analysis.

[3] In general it would be $\frac{h}{\ell}$ where h is the access stride/reuse vector length with $h < \ell$

5.1 Swim from SPEC 2000

`swim` is a scientific application. It has a main loop with four function calls. Each function has a loop nest for which the bounds are parameters introduced at run time. For sake of exposition, we present the analysis for the main loop nest of one procedure *calc*1() (Figure 3 written in C language). We analyze the interference for two different matrix sizes, the reference size 1335×1335 and the power of two 1024×1024. For the reference size, there is no interference for any cache line. For power of two matrices there is always interference.

The execution of SWIM with reference input takes 1hr on a sun ultra 5, 450MHz. Any full simulation takes at least 50 times more. Even the single loop simulation is time consuming. Our analysis takes less than one minute for each routine whether there is interference or there is no interference.

Due to the number of equations to verify, it is very difficult to verify by hand the accuracy of the analysis. We simulate 10 of the 800 calls to the *calc*1 routine using *cachesim5* from *Shade* [6]. The simulation results confirm our analysis.

5.2 Self Interference

We now consider self interference. Self interference happens when two references of the same array, or the same reference in different iterations, interfere in cache. The example, Figure 2, is the composition of six loops with only one memory reference in each. Each memory reference has a different spatial reuse and it is very long. STAMINA recognizes that the interval between reuses

Loop 0	Line	8	16	32	64	128	256
	$\epsilon_{ct}(L)$	0.50	0.25	1.00	1.00	1.00	1.00
Loop 1	$\epsilon_{ct}(L)$	0.50	0.25	0.12	1.00	1.00	1.00
Loop 2	$\epsilon_{ct}(L)$	0.50	0.25	0.12	0.06	1.00	1.00
Loop 3	$\epsilon_{ct}(L)$	0.50	0.25	0.12	0.06	0.03	1.00
Loop 4	$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	0.00
Loop 5	$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	0.00

Table 1. Self interference example. Loop four and five have no interference dependent from the line size, the output is set to zero

is after one iteration of the outer loop. It computes the reuse distance and, in the current implementation, it fixes the value of the interference density at $\rho = 1$. It assumes there is a miss due to capacity (in general the distance is not a constant and it cannot be compared to the cache size). For this particular case, it is a tight estimation. In general it is an over estimation. The existence of interference plays the main role, it discriminates when there is interference and when to count the interferences. In Table 1, we report the results of the analysis.

5.3 Tiling and Matrix Multiply

We analyze two variations of the common ijk-matrix-multiply algorithm (e.g. [10]). In Figure 4 the size of matrix A is not a power of two, but it is for B and C. The size of A has been chosen so that if there is interference due the reference on A, it does not happen very often. The index computation for A is parameterized ($0 \leq m \leq 64$ and $0 \leq n \leq 64$). Accesses on matrix C interfere with the accesses on B. Due to the upper bounds we choose for the parameters, A does not interfere with any other matrix. Even if it could, the interference density would be small. We are able to distinguish two different contribution: at compile time, $\epsilon_{ct}(L)$, and at run time, $\epsilon_{rt}(L)$. In Table 2 we can see that the suggested line

Line	8	16	32	64	128	256
$\epsilon_{ct}(L)$	2.00	1.00	2.00	2.00	2.00	2.00
$\epsilon_{rt}(L)$	0.00	0.00	0.00	0.00	0.00	0.00

Table 2. Interference table, for the procedure in Figure 4. Reference on A does not interfere with C and B with $0 \leq n, m \leq 64$. It would if we use lager parameters values. Note that the optimal line size is 16B. With a physical line of 32B the line size is optimal for both performance and energy.

size is 16B. This example has been introduced to show a case where the optimal line reduces interference and it is smaller than the common 32B line. Let us consider

Line	8	16	32	64	128	256
$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	0.00
$\epsilon_{rt}(L)$	2.00	2.00	2.00	2.00	2.01	2.03

Table 3. Interference table for the procedure *ijk_matrix_multiply_4* in Figure 5

a more interesting example, where we analyze the tiled version of matrix multiplication Figure 5. We analyze only the loop nest in the procedure *ijk_matrix_multiply_4*, and the result of the analysis is in Table 3. Every matrix interferes with every matrix. The interference due to matrix A is negligible since is an invariant for the inner loop. The interference between C and B can be at every iteration point. There is no interference whenever $|m - n| \mathbf{mod}\ \mathcal{C} = L$. This example is very peculiar because the line size is not set once for loop nest, it is determined at run time.

In the example in Figure 4 the analysis takes no more than two minutes. For the example in Figure 5 it takes more than 8 hrs, on a Sun ultra 5 450MHz. The difference of the execution times is expected. Most of the time is spent in the search for the existence of the integer solution. This is our performance bottleneck and it will be subject of further investigations/optimizations.

6 Summary and Future Work

We present a fast approach to statically determine the line size effect on the cache behavior of scientific applications. We use the static cache model introduced in [7] and we present an approach to analyze parameterized loop bounds and memory references. The approach is designed to investigate the trade-off between spatial reuse and interferences of loop nests on direct mapped cache. Experimental results demonstrate the accuracy and efficiency of our approach. We plan to expand our implementation to consider multi-way associative caches and to improve the performance of the existence test, by applying the *gcd*-test as proposed in [2].

7 Acknowledgment

The authors wish to thank Vincent Loechner, Sommath Ghosh and the members of AMRM project for their help on Ehrhart polynomials, the existence test, cache miss equation determination and our countless discussions. Financial support for this research was provided by DARPA/ITO under contract DABT63-98-C-0045.

References

1. E.Anderson, T.Van Vleet, L.Brown, J.Baer and A.R.Karlin "*On the Performance Potential of Dynamic Cache Line Sizes*". Technical Report UW-CSE-99-02-01.
2. Utpal Banerjee "*Loop Transformations for Restructuring Compilers The Foundations*". Kluwer Academic Publishers, January 1993
3. Ph. Clauss, "*Advances in parameterized linear diophantine equations for precise program analysis*", [ICPS RR 98-02], September 1998.
4. Ph. Clauss, V. Loechner, "*Parametric Analysis of Polyhedral Iteration Spaces*", research report ICPS 96-04, IEEE Int. Conf. on Application Specific Array Processors, ASAP'96, Chicago, Illinois, August 1996.
5. Ph. Clauss, "*Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs*", research report ICPS 96-03, 10th ACM Int. Conf. on Supercomputing, ICS'96, May 1996.
6. B. Cmelik and D. Keppel "*Shade: a fast instruction-set simulator for execution profiling*" Proceedings of the 1994 conference on Measurement and modeling of computer systems, 1994, Pages 128 - 137
7. S.Ghosh, M.Martonosi and S.Malik "*Cache Miss Equations: a Compiler Framework for Analyzing and Tunin Memory Behavior*" ACM Transactions on Programming Languages and Systems, Vol. 21, No. 4, July 1999, Pages 703-746.
8. K.Ghose and M.B.Kamble "*Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation*". Proceedings 1999 international symposium on Low power electronics and design, 1999, Pages 70 - 75
9. T.D.Givargis, J.Henkel and F.Vahid "em Interface and cache power exploration for core-based embedded system design". Proceeding of the 1999 international conference on Computer-aided design, 1999, Pages 270 - 273
10. G.H. Golub, C.F. Van Loan "*Matrix Computations*" Johns Hopkins Series in the Mathematical Sciences.
11. X.Ji, D.Nicolaescu, A.Veidembaum, A.Nicolau and R.Gupta "*Compiler-Directed Cache Assist Adaptivity*". ICS Techincal Report #00 17, May 2000.
12. M.B.Kamble and K.Ghose "*Analytical Energy Dissipation models for Low-power Caches*" Proceedings of the 1997 international symposium on Low power electronics and design, 1997, Pages 143 - 148
13. K.S.McKinley and O.Temam "*A Quantitative Analysis of Loop Nest Locality*". APLOS VII 10/96 MA, USA.
14. S.S. Muchnick "*Advanced compiler design implementation*". Morgan Kaufman.
15. S.J.E.Wilton and N.P.Jouppi "*CACTI: an Enhanced Cache Access and Cycle Time Model*". IEEE Journal of Solid-State Circuits. Vol. 31. No. 5, May 1996.
16. C.Su and A.M. Despain "*Cache design trade-offs for power and performance optimization: a case study* Proceedings 1995 international symposium on Low power design, 1995, Pages 63 - 68
17. . P.Van Vleet, E.Anderson, L.Brown, J.Baer and A.R.Karlin "*Pursuing the Performance Potential of Dynamic Cache Line Sizes*". Int. Conference on Computer Design (ICDD'99) October 1999.
18. A.V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau and X. Ji, "*Adaptive Cache Line Size to Application Behavior*", In Proceedings of International Conference on Supercomputing (ICS). June 1999, pp.145-154.
19. X. Vera, J. Llosa, A. Gonzales and N. Bermudo, "*A Fast and Accurate Approach to Analyze Cache Memory Behavior*" EUROPAR 2000
20. D.K. Wilde "*A library for Doing Polyhedral Operations*". Publication interne N 785, 1993
21. M.E.Wolf and M.S.Lam *A data Locality Optimizing algorithm* Proc. of the ACM SIGPLAN'91 Conference on programming languages design and implementation, Toronto, Ontario, Canada, June 26-28, 1991, pages 30-44.

```
#define CACHE_SIZE 16384

int A[CACHE_SIZE /16][(CACHE_SIZE+16)/4];
int B[CACHE_SIZE / 32][(CACHE_SIZE+32)/4];
int C[CACHE_SIZE / 64][(CACHE_SIZE+64)/4];
int D[CACHE_SIZE / 128][(CACHE_SIZE+128)/4];
int E[CACHE_SIZE / 256][(CACHE_SIZE+256)/4];
int F[CACHE_SIZE / 512][(CACHE_SIZE+512)/4];


int
main ()
{
  int i,j,k,l;
  int step;
  l = 0;


  for (j=0;j<4;j++) {
    for (k = 0; k < CACHE_SIZE / 16 k++)
      A[k][j]++;
  }

  for (j=0;j<8;j++) {
    for (k = 0; k < CACHE_SIZE / 32; k++)
      B[k][j]++;
  }

  for (j=0;j<16;j++) {
    for (k = 0; k < CACHE_SIZE / 64; k++)
      C[k][j]++;
  }


  for (j=0;j<32;j++) {
    for (k = 0; k < CACHE_SIZE / 128; k++)
      D[k][j]++;

  }

  for (j=0;j<64;j++) {
    for (k = 0; k < CACHE_SIZE / 256; k++)
      E[k][j]++;
  }

  for (j=0;j<128;j++) {
    for (k = 0; k < CACHE_SIZE / 512; k++)
      F[k][j]++;
  }

  return 0;
}
```

Fig. 2. Self Interference and analysis results

```
#define N1 1335
#define N2 1335

extern  double U[N1][N2], V[N1][N2], P[N1][N2],UNEW[N1][N2], VNEW[N1][N2],
        PNEW[N1][N2], UOLD[N1][N2],  VOLD[N1][N2], POLD[N1][N2],
        CU[N1][N2], CV[N1][N2], Z[N1][N2], H[N1][N2], PSI[N1][N2];
extern double D0, DX, DY;

void calc1(int M, int N) {
  int i,j;
  double FSDX,FSDY;

  for (i=0;i<M;i++)
    for (j=0;j<N;j++) {

      //     RN 0         =        1        2       3
      CU[i+1][j] = D0*(P[i+1][j]+P[i][j])*U[i+1][j];
      //C     #  1 2 3 0

      //C     RN 4                 5        2       6
      CV[i][j+1] = D0*(P[i][j+1]+P[i][j])*V[i][j+1];
      //C     # 5 2  6 4
      //C     RN 7                     8          6             9
      Z[i+1][j+1] = (FSDX*(V[i+1][j+1]-V[i][j+1])-FSDY*(U[i+1][j+1]
      /* C     RN       3            2      1          10         5   */
                   -U[i+1][j]))/(P[i][j]+P[i+1][j]+P[i+1][j+1]+P[i][j+1]);
      //     # 8 6 9 3 2 1 10 5 7
      //        RN 11       2             3         3         12     12
      H[i][j] = P[i][j]+D0*(U[i+1][j]*U[i+1][j]+U[i][j]*U[i][j]
     //     RN       9         9        13      13
                         +V[i][j+1]*V[i][j+1]+V[i][j]*V[i][j]);
      //     #  3 12  9 13 2 11
    }

}
```

Fig. 3. SWIM: calc1() in C code, in the comment lines the reference number and the order of the references are specified.

```
/* B[0][0] 120000000
   C[0][0]
*/
#define MAX 4000
#define MAXCOL 2048


double A[MAX][MAX],  B[MAXCOL][MAXCOL],  C[MAXCOL][MAXCOL];
void ijk_matrix_multiply( int n, int m) {

  int i,j,k;


  for(i=0;i<n;i++)
    for(k=0;k<n;k++)
      for(j=0;j<n;j++)
        C[i][j+3] += A[i][k+m] * B[k][j];


}
```

Fig. 4. Matrix Multiply. Two parameters: loop bounds and A offset. The parameters n and m up to 64

```
#define MAX 2048
double A[MAX][MAX],  B[MAX][MAX],  C[MAX][MAX];

void ijk_matrix_multiply_4( int x,int y, int z, int m, int n, int p ) {
  int i,j,k;

  for(i=0;i<x;i++)
    for(k=0;k<y;k++)
      for(j=0;j<z;j++) {
        C[i][j+m] += A[i][k+n] * B[k][j+p];

}


void matrix_multiply_new_tiling() {
  int ii,jj,kk;

  for(kk=0;k<MAX/b;kk++)
    for(ii=0;i<MAX/b;ii++)
      for(jj=0;j<MAX/b;jj++)
        ijk_matrix_multiply_4(min(b,MAX-ii*b), min(b,MAX-jj*b),
                              min(b,MAX-kk*b), (ii*MAX+jj)*b,
                              (ii*MAX+kk)*b,(kk*MAX+jj));
}
```

Fig. 5. Tiling of Matrix Multiply. 6 parameters: loop bounds and A,B and C offsets. The first procedure describes the computation on a tile.