The Pennsylvania State University
The Graduate School
College of Engineering

REACTING TO OS/SYSTEM RESOURCE PARTITIONING DECISIONS: TILING EXAMPLE

A Thesis in
Computer Science and Engineering
by
Jithendra Srinivas

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2011

The thesis of Jithendra Srinivas was reviewed and approved* by the following:

Mahmut Kandemir
Professor of Computer Science and Engineering
Thesis Advisor

Chita Das
Distinguished Professor of Computer Science and Engineering
Professor

Raj Acharya
Professor of Computer Science and Engineering
Deaprtment Head

*Signatures are on file in the Graduate School.

Abstract

To fully exploit emerging multicore architectures, managing shared resources (i.e., caches) across applications and over time becomes critical. All prior efforts view this problem from the OS/system side, and do not consider whether applications can participate in this process. In this paper, we show how the application can react to OS/system resource management decisions by adapting itself, with the objective of maximizing the utilization of shared resources allocated to it. Specifically, our reactive-tiling strategy enables applications to react to OS/system resource allocation decisions. Experimental results show that our scheme is very effective in practice.

Table of Contents

List of Figures **v**

Acknowledgments **vi**

Chapter 1
Introduction **1**

Chapter 2
The Polyhedral Model **4**

Chapter 3
Framework for Reactive Tiling **7**
3.1 High-Level Operation 7
3.2 Code Generation for Multiple Tile Sizes 9
3.3 Safe Point Analysis 10
3.4 Graphical Illustration of Safe Points 12
3.5 Code Unification 13

Chapter 4
Experimental Setup and Results **15**
4.1 Experimental Setup 15
4.2 Experimental Results 17

Chapter 5
Related Work **19**

Chapter 6
Concluding Remarks **21**

Bibliography **22**

List of Figures

1.1 Variation of tile size with cache allocation. X-axis shows the tile size that generates the best results while Y-axis shows the corresponding OS-based cache allocation to an application. Here we assume other tiling parameters such as tile shape as fixed. 2

3.1 Illustration of dynamic application adaptation to OS cache allocations at runtime. Note that there is a time gap between the new OS allocation and application's reaction to it. The application switches to the new tile only at "safe points". . . 7
3.2 A high level overview of the code transformation phases. 8
3.3 Transformed codes on different phases in reactive tiling framework. (a) is the original code; (b) and (c) are the tiled codes with tile sizes $16 \times 32 \times 16$ and $32 \times 64 \times 32$, respectively. These constitute the output of the code generation phase; (d) and (e) represent the codes where safe points are inserted. They are the output of the safe point analysis phase; (f) is the code after the code unification step. 9
3.4 Graphical illustration of safe points with different tile sizes. (a) and (b) illustrate the tiled iteration spaces with tile sizes 2×2 and 4×4, respectively. Black dots represents the iteration points; the safe points i_1 and i_2 are cycled; arrows indicate the lexicographic execution order of the iterations. 12

4.1 Scenarios of the cache allocation variations. The x-axis and y-axis in all the nine Scenarios represent the time and cache size, respectively. 16
4.2 Results from the synthetic allocation cache allocations 17
4.3 Results from utility based cache allocations 18

Acknowledgments

First and foremost, I would like to express my sincerest gratitude towards my advisor, Mahmut Kandemir. His critical remarks and judgement motivated me to think in new directions. I would like to thank my collaborators in this work: Shekhar Srikantaiah, Wei Ding, Yang Ding, Hui Zhao and Akbar Sharifi. Without their input it would not have been possible to build the final system. I'd like to thank Madhav Jha for helping me in writing the thesis.

Chapter 1

Introduction

Due to increasingly problematic effects of clock frequency on power consumption and heat generation, there is a shift in chip manufacturing from complex single core machines to simple multicore architectures. While this move helps with power and temperature related issues and holds the complexity of a single core somewhat static over time, it also brings its own set of problems. First, using these architectures requires parallelizing single-threaded applications. Second, increasing core counts and limited off-chip bandwidth can result in pressure on communication bandwidth and memory accesses, respectively. Third, it is not clear how system software should be structured/redesigned for these architectures. Despite these challenges, the chip manufacturers such as IBM, Sun, Intel and AMD already have multicore products in the market [9, 10, 11, 12], and one can expect these emerging architectures to be the building blocks of any future computer system from smart phones to laptops to desktops to supercomputers. It has been projected that future multicores will have several interesting characteristics, as pointed out in [9, 10, 11, 12].

One of the important characteristics of emerging multicore machines is large number and variety of *shared resources* [9, 10, 11, 12]. A typical multicore system employs several shared resources such as on-chip caches, on-chip network, processor cores and off-chip bandwidth. How these resources are managed across applications and over time influences system performance significantly. For instance, recent research clearly demonstrates that on-chip shared cache (L2 or L3) management is critical for application/workload performance [35], [15] , [3]. These approaches typically modulate cache space allocated to an application based on a predefined objective function (e.g., weighted speedup [1] and/or fairness [15]).

All prior efforts look at this cache partitioning problem from the OS/system side, and they do not explore the possibility of whether applications can also play a role in this process. This is unfortunate because the OS typically partitions a shared cache space at runtime based on some global (workload wide) metric (e.g., weighted speedup) among the applications in a workload, and as a result of this partitioning, performance of a given application in this workload can suffer dramatically.

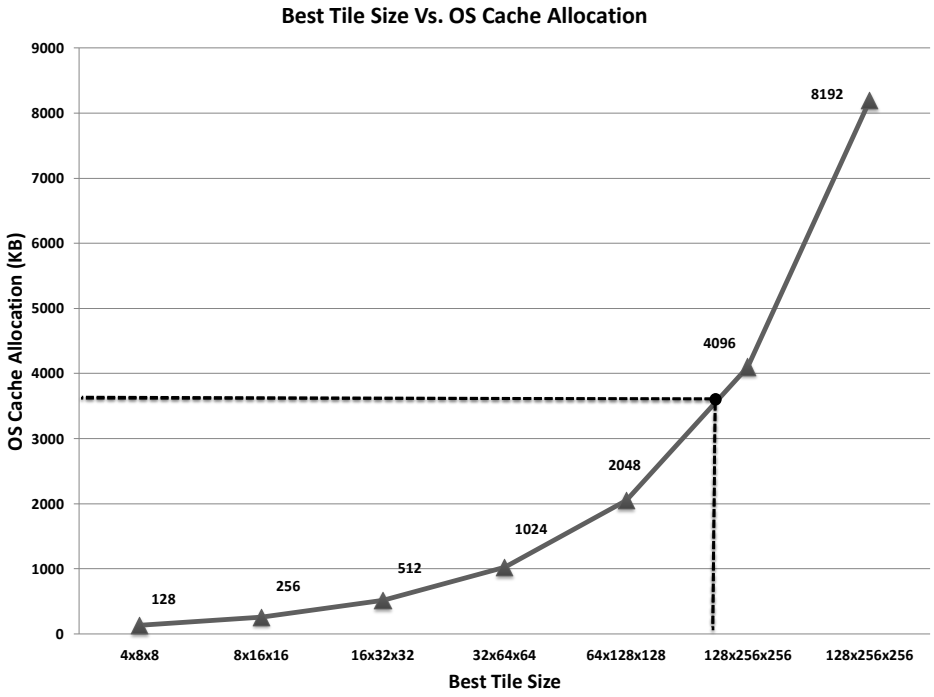


Figure 1.1. Variation of tile size with cache allocation. X-axis shows the tile size that generates the best results while Y-axis shows the corresponding OS-based cache allocation to an application. Here we assume other tiling parameters such as tile shape as fixed.

Motivated by this, we investigate a novel concept in this work called *reactive application*. Specifically, a reactive application is the one that can react to OS/system based resource management / partitioning decisions by adapting itself. As a specific instance of reactive application, we consider a dynamic version of *iteration space tiling* (also called *loop blocking*), a well-known compiler optimization, to adapt to variations in available cache space. Iteration space tiling [24, 33, 36, 37, 40, 18] partitions iteration space of a loop nest into smaller chunks (blocks), so as to help ensure that data reuses can be satisfied from cache memory. An important parameter in tiling is the *tile size* (also known as the *blocking factor*), which determines the chunks of data blocks accessed at any given time. It has been shown that by prior research [16, 20, 17, 18, 23, 19] that tile size is a critical parameter that determines overall performance of a loop nest. As shown in Figure 1.1, ideally, tile size selection should be based on the available cache capacity to the application. Our point is that, when the OS changes the cache allocation to an application at runtime, a tiled application can *react* to this move by changing it tile size. We want to emphasize however changing tile size at runtime is not trivial as we need to (i) decide what tile size to use next and (ii) determine a suitable program point at which the switch should occur.

One can expect two potential benefits from this approach. First, matching tile size to available cache capacity dynamically (during execution) improves performance of the target application. And second, better utilization of cache space reduces pressure on other applications (co-runners)

that execute concurrently with the target application. The reactive application, by adapting to a lower cache allocation, mitigates the pressure on co-running applications by making more cache space available to them, without incurring significant loss in quality-of-service (QoS) or global fairness metrics. We implemented our approach in a compiler framework and performed experiments with different execution scenarios. The specific contributions of this work can be summarized as follows:

- To our knowledge, this is the first work that considers application reaction to system/OS induced cache allocations.
- We present a framework that can generate code for adaptive (reactive) tiling. In this approach, the generated code accommodates a set of *safe points* at which one can switch from one tile size to another (to react to dynamic OS allocations).
- We explain an execution model in which a reactive application (i.e., an application with adaptive tiling) can react to the modulations in its cache space allocations to prevent its performance from degrading significantly.
- We present experimental evidence showing that our proposed approach works well in practice. Specifically, with synthetic cache allocations reactive tiling improves performance of applications by 19.2% (on average), and with dynamic cache allocations using utility based cache partitioning improves performance of applications by 10.5% (on average) when compared to the scenario where applications do not react to OS-based modulations in cache allocation.

The remainder of this thesis is structured as follows. The next chapter explains the basics of the polyhedral model and tiling. Chapter 3 presents the details of our proposed reactive tiling strategy. Chapter 4 gives an experimental evaluation, Chapter 5 discusses related work and the thesis is concluded in Chapter 6.

Chapter 2

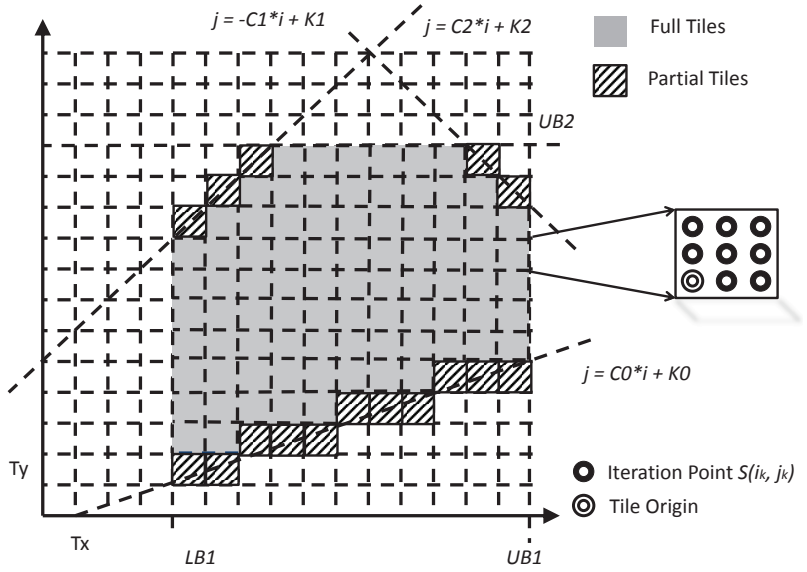
The Polyhedral Model

```
/* Original loop nest before tiling*/
for i=LB1 to UB1 do
  for j=C0*i+K0 to min(-C1*i+K1, C2*i+K2, UB2) do
    S(i, j);
```

(a) Code before tiling transformation

```
/* Outer loop indices enumerate the origin of a tile*/
for ii= ⌊LB1/Tx⌋ to ⌊UB1/Ty⌋ do
  for jj=C0*i+K0 to min(-C1*i+K1, C2*i+K2, UB2) do
    /* Inner loop indices i and j scan each statement */
    for i=max(LB1, ii*Tx) to min(UB1, ii*Tx + Tx − 1) do
      for j=max(C0*i+K0, jj*Ty) to min(jj*Ty + Ty − 1, min(-C1*i+K1,
      UB2, C2*i+K2)) do
        S(i, j);
```

(b) Code after tiling transformation



(c) Illustration of the polyhedral model

In this chapter, we first introduce the polyhedral model, the basis of the existing tiling

techniques. In its most general form, tiling decomposes an n-dimensional loop nest into a $2n$-dimensional loop nest where the outer n loops iterate over the tiles and the inner loops iterate over the points within a tile. Our focus is on the loop nests whose bounds and array references are affine functions of loop indices and other global parameters (e.g., input size).

In the polyhedral model, an n level loop nest represents an n-dimensional *iteration space* I, and each iteration can be expressed by an iteration vector $\vec{i} = (i_1, i_2, \cdots, i_n)^T$, where i_k is the index of the k-th loop (starting from the outermost one). Each i_k satisfies the boundary constraints $L_k \leq i_k \leq U_k$, where L_k and U_k are the corresponding lower and upper loop bounds, respectively. For a statement s within such a loop nest, the set of iterations for which s has to be executed can always be specified by a set of affine linear inequalities that are derived from loop indices. These inequalities define an *iteration space polytope* in which a dynamic instance (iteration) of each statement is represented as an integer point (expressed as its *iteration vector*).

With such a representation for each statement, it is easy to capture the dependence (inter- and intra- statement) within the iteration space polytope and reason about the correctness of loop transformations. An instance of statement s (denoted as $\vec{i_s}$) depends on an instance of statement t (denoted as $\vec{i_t}$), if they access the same memory location and $\vec{i_s}$ is executed before $\vec{i_s}$ within the valid iteration space polytope. This can be expressed as:

$$(\vec{i_s}, \vec{N}, 1)^T = H.(\vec{i_t}, \vec{N}, 1)^T, \tag{2.1}$$

where $\vec{N}$ is the vector includes all the global parameters and "1" indicates the offset. H is called the *transformation matrix* which preserves the dependence between s and t. The left hand side and right hand side in Equation 2.1 refer to the logical memory locations accessed by statements s and t, respectively. To ensure the correctness of loop transformation, this dependence must be preserved in the transformed loop, which indicates the execution order between s and t. On the other hand, generating the transformed loop in the polyhedral model can be considered as specifying an (execution) order to visit each integral point in the iteration space polytope (known as *scanning* the polyhedra [38]), *once and exactly once*. Therefore, how to determine such an execution order is extremely important for the correctness of loop transformation. More on this will be discussed in Chapter 3.2.

Tiling is a special type of loop transformation (restructuring). Existing tiling schemes [29, 30, 31, 32, 27, 24] optimize for data locality and parallelization by reordering the execution of the statements in the loop body. The constraints described in the previous paragraph guarantee the correctness of such program execution reordering. When tiling is performed, in the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving *supernode or origin* iterators and intra-tile iterators, which specify the execution order of inter- and intra-tiles, respectively. Figure 2(b) illustrates the tiled version of the code shown in Figure 2(a). In this example, $(ii, jj)^T$ enumerates a supernode tile. The process of enumerating supernodes is referred to as *inter-tile scanning* and that of enumerating the points inside the tile is referred to as *intra-tile scanning*. It is important to note that our proposed reactive tiling

framework is sensitive to the inter-tile scanning.

Figure 2(c) illustrates this transformation visually. When tiling is applied, the original two-dimensional loop nest in this example is transformed to a four-dimensional loop nest. The axes represent the loop iterators i and j, the shaded region represents the iteration points inside the polyhedra, and the boundaries of the shaded region represent the loop bounds (affine). Observe that the polyhedra is segmented into blocks, which represent the tiles in the tiled iteration space. Further, the iteration space consists of *partial tiles* and *full tiles*. We can distinguish a full tile from a partial tile based on the inclusion of iteration points in the tile. In a full tile, all the iteration points are scanned during the execution of the loop nest, whereas, in a partial tile, only a subset of the iteration points are scanned during the execution of the loop nest. This disparity arises from the fact that the boundaries of the loop nest may not coincide exactly with the tile boundaries; intra-tile scanning ensures that only the iteration points in the partial tile which are interior to the polyhedra are scanned during the execution of the program. Our proposed tiling scheme does not only handle the case where only full tiles exist in the transformed loop nest, but also, the case where full tiles and partial tiles co-exist (more explanation in Chapter 3.3).

Chapter 3

Framework for Reactive Tiling

3.1 High-Level Operation

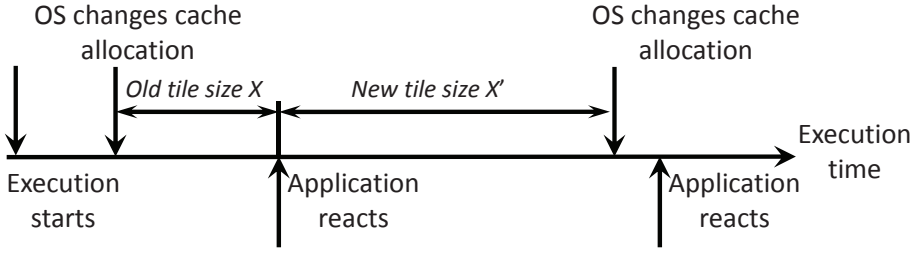


Figure 3.1. Illustration of dynamic application adaptation to OS cache allocations at runtime. Note that there is a time gap between the new OS allocation and application's reaction to it. The application switches to the new tile only at "safe points".

Figure 3.1 outlines the high level operation of our reactive tiling strategy. The horizontal line represents application execution timeline. At the beginning of the application execution (shown on the left end of the timeline), the tile size is set to X [1]. After executing a few loop iterations, the loop nest reaches a *safe point*, at this point the program switches to the tile size X' (if necessary), which is predicted to generate the best performance. After a while, the OS changes the cache allocation again, the application continues execution with the previous tile size till it reaches another safe point. The application reacts again be performing another adaptation by switching into a potentially new tile size. This process is repeated until the program terminates. The technical properties of our safe point identification strategy are discussed in Chapter 3.3.

An important question at this point is how the tile size for a given cache allocation is determined. While different methods are possible to do this, in our base implementation, we use a *profile based* offline method. In this method, an application is profiled (under different cache

[1] Initial tile size selection can be made based on assuming a specific cache allocation

sizes), and for each cache size tested, we determine the tile size that generates the best result. This can be achieved by executing a few tiles using controlled cache allocation and selecting the best tile size from the available options. In this way, we obtain a two-dimensional curve where the x-axis represents OS-specified cache allocations and the y-axis shows the corresponding tile sizes (Figure 2 capture for the contents of such a curve). Note however that it is possible during execution to observe a cache allocation for which we do not have profile data. To address this issue, we employ *curve-fitting*. More specifically, we determine the best tile sizes for the cache allocations for which we do not have experimental data using curve-fitting (which in a sense corresponds to a refinement of the initial curve we have). Further, at runtime, when we run the application with a tile size for which we do not have a point in our curve, we update the curve (with the performance data corresponding to that tile size), i.e., we invoke curve-fitting to have a more accurate tile size-performance model.

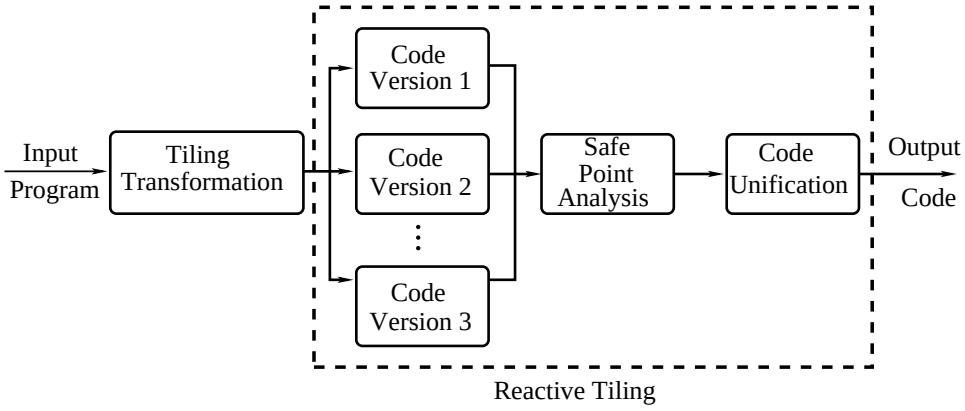


Figure 3.2. A high level overview of the code transformation phases.

Figure 4 illustrates the main components of the compiler part of our proposed approach. Our source-to-source transformation tool takes a user-provided sequential code as input. We employ Pluto [27], a loop transformation tool, to perform the necessary code restructuring (though other tools can also be used for this purpose). The tiling transformation reorders statement execution in the transformed iteration space, preserving the original semantics of the program. Our framework for reactive tiling is enclosed in the bounded box in Figure 4.1. The transformed code with multiple instances of tile sizes can be considered as the preliminary step for our Safe Points Analysis and Code Unification. Safe Points are specific points in the program execution which ensure correctness of execution when the program switches from one tile size to another. Code Unification on the other hand is essentially a code generation phase which produces a unified code by merging multiple tile sizes which can adapt themselves to changing system resources during runtime. Finally, we generate, as output, a tiled code with the appropriate tile sizes that maximize performance and utilization of shared cache space allocation. Basics of tiling transformation are already discussed in Chapter 2; forthcoming chapters elaborate more on Code Generation, Safe Point Analysis, and Code Unification phases.

3.2 Code Generation for Multiple Tile Sizes

Tiling transformation (explained in Chapter 2) is followed by the code generation phase. Currently, we employ CLooG code generator [28] to implement our code generation. However, if desired, one can easily replace it with any other existing code generator.

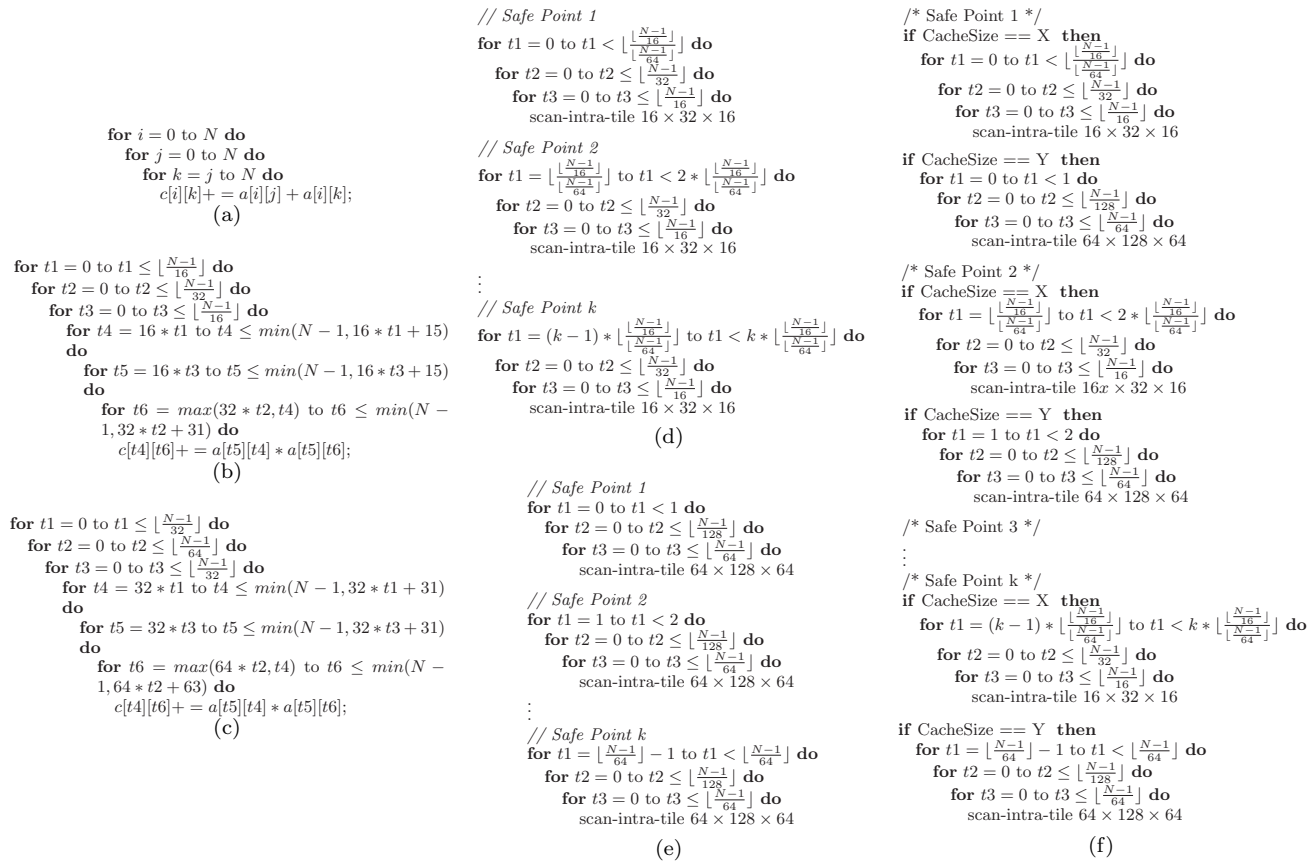


Figure 3.3. Transformed codes on different phases in reactive tiling framework. (a) is the original code; (b) and (c) are the tiled codes with tile sizes $16 \times 32 \times 16$ and $32 \times 64 \times 32$, respectively. These constitute the output of the code generation phase; (d) and (e) represent the codes where safe points are inserted. They are the output of the safe point analysis phase; (f) is the code after the code unification step.

Recall that the generated tiled code can be viewed as scanning the integral iteration points inside the polyhedra under a specified lexicographic ordering, once and only once. Meanwhile, the dependences (if exist) in the original program also need to be preserved. In CLooG, this order can be characterized by the following affine function (called the *scattering function*):

$$\phi(\vec{i}) = C.(\vec{i}, \vec{N}, 1)^T, \tag{3.1}$$

where C is a constant matrix, $\vec{N}$ is the vector includes all the global parameters and "1" indicates the offset. $\phi(.)$ represents the logical execution time for the iteration $\vec{i}$. If an instance of statement

s (denoted as $\vec{i_s}$) depends on an instance of statement t (denoted as $\vec{i_t}$), then $\vec{i_s}$ should be executed earlier than $\vec{i_t}$. Therefore, we have $\phi(\vec{i_s}) \prec \phi(\vec{i_t})$.[2] As a result, the key to scanning the polyhedra is to determine the matrix C. More discussion on C can be found elsewhere [26]. In CLooG, Quillere's algorithm [25] is used since it gives the best results when generating code for several polyhedra.

During the code generation phase, a tiled code that can accommodate multiple tile sizes is generated. The required tile sizes for code generation are determined using application profiling and regression methods; the set of tile sizes that should be used for code generation will be explained in detail in the experiments chapter. Figure 3.3(b) and Figure 3.3(c) represent the generated tiled codes with tile sizes $16 \times 32 \times 16$ and $32 \times 64 \times 32$, respectively, for the original code given in Figure 3.3(a).

We generate tiled code with multiple tile sizes for two purposes. Firstly, during an application execution, modulations in shared cache allocations can result in application performance variations. A fixed tile size may not give the best performance when the amount of available shared cache to an application varies. It can be further argued that varying the tile size in accordance with the shared cache allocation can result in the best application performance. Decisions regarding the choice of the right tile size, which gives best performance improvement, need to be made at runtime. Secondly, in an environment with dynamic changes in the amount of shared cache allocated to an application, our proposed reactive tiling scheme will switch to the right tile size at runtime. Generating multiple tile sizes assists in the decision of choosing appropriate tile sizes.

3.3 Safe Point Analysis

In this chapter, we introduce the concept of *safe point*. Recall that previous chapter discussed the code generation for different tile sizes. Even though the code was generated for different tile sizes, program semantics are not affected. However, selecting the right tile size can affect the performance of the application. Optimal tile size selection problem has been studied in the context of auto-tuning programs in which the program parameters such as tile sizes are determined based on iterative techqniues [21].

The objective of safe point is to provide a seamless mechanism that enables the code to switch from one tile size to another at runtime without affecting the correctness of the program. The distinguishing feature of our method is that a tiled application can switch from one tile size to another while it is running. Unlike existing approaches, such as [22] which requires the program to be re-run from the beginning, our approach ensures that the current state of the computation is *preserved* while making a switch from one tile size to another. Shortly later, we define safe points and give an algorithm on how to compute safe points given multiple tile sizes.

[2]Lexicographic ordering: Consider vectors $\vec{a} = (a_1, a_2, \cdots, a_n)$ and $\vec{b} = (b_1, b_2, \cdots b_n)$ in a n-dimensional space. $\vec{a}$ is lexicographically smaller than $\vec{b}$, denoted by $\vec{a} \prec \vec{b}$ if $a_1 < b_1$ or both $a_1 = b_1$ and $(a_2, \cdots a_n) \prec b_2, \cdots b_n)$. Similarly, $\vec{a} \preceq \vec{b}$ if $a_1 \leq b_1$ or both $a_1 = b_1$ and $(a_2, \cdots, a_n \preceq b_2, \cdots, b_n)$.

Algorithm 1 Safe Points Analysis Algorithm

Input: (N_1, T_1), (N_2, T_2), $\cdots$, (N_n, T_n) pairs, where N_i is the total number of tiles of the (same) loop nest with tile size T_i

Output: $S_{T_1}, S_{T_2}, \cdots, S_{T_n}$, where S_{T_i}, denotes the set of safe points for tile size T_i.

```
1: begin:
2: if L.C.M(T_1, T_2, ..., T_n) < min(N_1, N_2, ..., N_n) then
3:   for i = 1 to n do
4:     λ_i ← LCM(T_1, T_2, ..., T_n)/T_i;
5:     i ← 0;
6:     safePointVar ← 0;
7:     S_{T_1} ← null;
8:     /*for each λ_i, calculate all the safe points */
9:     while safePointVar < min(N_1, N_2, ..., N_n) do
10:      safePointVar ← i × λ_i;
11:      S_{T_i} ← S_{T_i} ∪ safePointVar;
12:      i ← i + 1;
13:    end while
14:  end for
15: else
16:   No safe point exists.
17: end if
18: return S_{T_1}, S_{T_2}, ..., S_{T_n}.
19: end
```

Local Safe Point: Let T_1 and T_2 be two tile sizes of the same loop nest. Then, the origin of every λ_1 tile (with respect to tile size T_1) and the origin of every λ_2 tile (with respect to tile size T_2) are common to both the tiles. These origins are called *local safe points*. Specifically, λ_1 = LCM$(T_1, T_2)/T_1$ and λ_2 = LCM$(T_1, T_2)/T_2$.[3]

Global Safe Point: Let $T_1, T_2, \cdots, T_n$ be the tile sizes of n different loop nests. Without loss of generality, the origin of every λ_i tile (with respect to tile size T_i) is referred as the *global safe point*, where $1 \leq i \leq n$ and λ_i = LCM$(T_1, T_2, \cdots, T_n)/T_i$.

The local safe points can be used for switching between two tile sizes in the same loop nest. In constrast, the global safe points can be used for switching between tile sizes across multiple loop nest. Generally speaking, there are much fewer global safe points than local safe points according to the above definitions. It highly depends on the current context to decide what kind of safe points should be accommodated. One can find that the local safe point is actually a special case of global safe point. Let $code_1$, $code_2$, $\cdots$, $code_n$ be code versions generated at the end of code generation phase. By definition, local safe points can be used to swtich between two code versions, $code_i$ and $code_j$, however, if the code has to be switched from one version to any

[3]LCM denotes the mathematical Least Common Multiple function.

of the n other code versions, only a global safe point can be used. In the rest of this thesis, for simplicity, we will use the term *safe point* in referring to both of them.

Lemma: Given a set of n pairs: (N_1, T_1), (N_2, T_2), $\cdots$, (N_n, T_n), where N_i is the total number of tiles for a loop nest with tile size T_i, a safe point exists iff LCM$(T_1, T_2, \cdots, T_n) < min(N_1, N_2, \cdots, N_n)$.

Proof. Based on the definition of safe point, after executing λ_i number of tiles with tile size T_i $(1 \leq i \leq n)$, a safe point is reached. Since LCM$(T_1, T_2, \cdots, T_n) < min(N_1, N_2, \cdots, N_n)$, a safe point is reached before the completion of execution of tiles in either loop nests. Hence, a safe point is guaranteed to exist □ □

The algorithm for calculating safe points is given as Algorithm 1 below. This algorithm takes as input the tile sizes generated from the tile generation phase. If LCM$(T_1, T_2, \cdots, T_n)$ is greater than $min(N_1, N_2, \cdots, N_n)$, then a common safe point does not exist. Otherwise, we generate all the safe points for each given tile size (lines 9 through 13).

3.4 Graphical Illustration of Safe Points

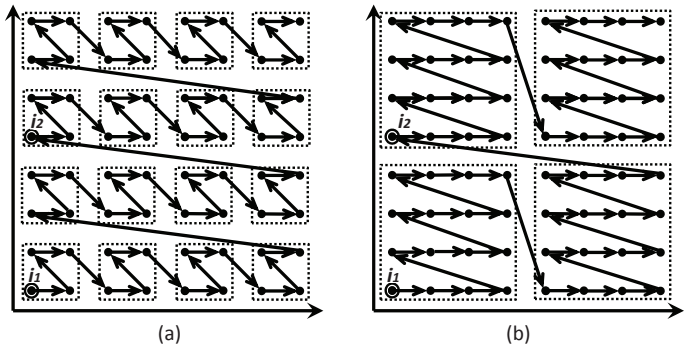


Figure 3.4. Graphical illustration of safe points with different tile sizes. (a) and (b) illustrate the tiled iteration spaces with tile sizes 2×2 and 4×4, respectively. Black dots represents the iteration points; the safe points i_1 and i_2 are cycled; arrows indicate the lexicographic execution order of the iterations.

For simplicity, we use a two-dimensional loop nest illustrating the safe points. Figures 3.4(a) and 3.4(b) correspond to the same loop nest with different tile sizes, T_1 and T_2, respectively. In these figures, the loop iterators form the co-ordinate axes. And the black dots represent the iterations and the safe points are cycled. An arrow indicates the lexicographic execution order of the iterations. In this example, Figure 3.4(b) represents a tile size larger than Figure 3.4(a), i.e., $T_2 > T_1$, which explains the reason for fewer tile origins in Figure 3.4(b). Based on the definition in Chapter 3.3, safe points i_1 and i_2 are calculated for both tile sizes T_1 and T_2. Specifically,

Algorithm 2 Code Unification Algorithm

Input: T_1, T_2, $\cdots$, T_n; S_{T_1}, S_{T_2}, $\cdots$, S_{T_n}; Original Code
Output: Unified Tile Code

```
1: for each tile T_i ∈ (T_1, T_2, ··· , T_n) do
2:   for each safe point sp ∈ S_{T_i} do
3:     Unroll and Compute the loop bounds at each safe point
4:     Identify best cache size X_i for tile size T_i
5:     Insert predicate if(CacheSize == X_i)
6:     Terminate the predicate at the end of safe point
7:   end for
8:   Merge code between safe points;
9: end for
```

we have $S_{T_1} = \{0, 8\}$ and $S_{T_1} = \{\ 0,\ 2\ \}$. At any of them, a transistioning can be made from one tile size to another without affecting the correctness of the execution of the program. The corresponding λ_1 and λ_2 are 8 and 2, respectively.

A loop nest example comparing safe points for two tile sizes is shown in Figures 3.3(d) and 3.3(e). In this example, the code on the left hand side corresponds to a tile size of $T_1 = 16 \times 32 \times 16$ and the code on the right hand side corresponds to a tile size of $T_2 = 64 \times 128 \times 64$. This indicates that we have $T_2 = 64 \times T_1$. Therefore, by using the lemma defined earlier, for every execution of 64 smaller tiles, the tile origins are overlapped, and the safe points are formed. The original loop nest is now unrolled at these safe points and the new loop bounds are computed. The take away here from this discussion is that, during program execution, the code can switch from the safe points defined in one loop nest to corresponding safe points of the other loop nest, thus effectively changing the tile size at runtime safely without affecting the correctness of the program. The next chapter explains a code unification mechanism that produces a unified code for multiple tile sizes which adapts itself (or auto-tunes itself) to varying cache allocations.

3.5 Code Unification

The safe point analysis algorithm shown in Algorithm 1 returns the list of the safe points for given input tile sizes. We use these safe points during the code unification step. Code unification is performed in two phases, the first phase is the unroll phase and the second phase is merge. During the unroll phase, nests are unrolled up to the safe points, and new loop bounds for the unrolled loop nests are calculated. Once the unroll phase is complete, we proceed to the merge phase. During the merge phase, an appropriate tile size selection is carried out based on the shared cache allocation. We introduce a predicate which checks the state of the current cache allocation at a safe point and, based on the cache allocation selects the right tile size. Our code unification strategy is given in a pseudo-code form in Algorithm 2.

An example for code unification for two tile sizes is given in Figure 3.3(f). In this example two cache allocations are studied, X KB and Y KB. We used the symbolic variable CacheSize to inform the application of the varying cache size. In practice this can be achieved through either

the use of a general purpose register or a signaling mechanism. At a safe point, an application can check the variable CacheSize and if the value of CacheSize is X, then the application chooses the tile size of $16 \times 32 \times 16$, and if the value of CacheSize is Y, then the application chooses the tile size of $64 \times 128 \times 64$. The details are explained in the next chapter.

Chapter 4

Experimental Setup and Results

In this chapter, we present a detailed description of experimental setup and results. For clarity our results are segregated into two classes. The first class consists of different synthetic cache allocations; and second class consists of reactive application co-running with SPEC2006 benchmarks, and we use utility based cache partitioning [35]

4.1 Experimental Setup

We evaluated our reactive tiling scheme on a 4-core CMP, with cores based on Linux X86 architecture. The relevant details of this architecture are presented in Table 1. The application is initially profiled with various tile sizes and shared cache allocations. This profiling determines the right tile size for a given shared cache allocation. The results from profiling will be used later to identify the best tile size from a pool of tile sizes for a given cache configuration. All the experiments are performed using two different versions explained below.

• *Default.* In this version, the application uses a *static tile size*. In this configuration, the application is oblivious to the changing system cache allocations. Before the execution begins, the application checks with the system the amount of cache space allocated to it. Based on the amount of shared cache available to the application, the best tile size from the given pool of tile sizes is determined. After selecting the best tile size from the available pool of tile sizes, the application adheres to the fixed tile size until it terminates.

• *Reactive.* In this version, multiple tile sizes can be exercised during the course of execution. The application tracks the modulations in its cache allocation. Similar to the default version, before the execution, the application chooses the best tile size from the pool of tile sizes available. However, the application will no longer adhere to the initial tile size chosen. Specifically, when it reaches a safe point during its execution, it checks the amount of shared cache available to it at that point. If the application notices a change in the available cache allocation, it dynamically switches to the best tile size for the new shared cache allocation.

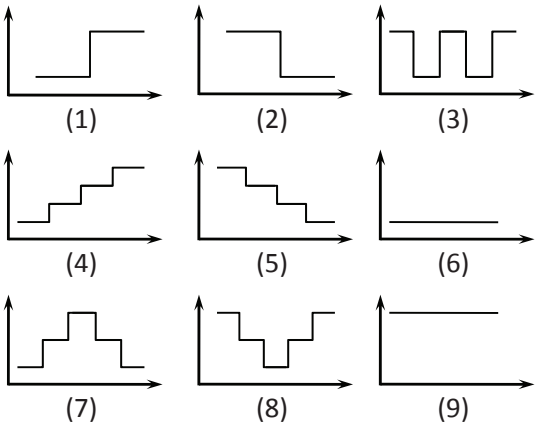


Figure 4.1. Scenarios of the cache allocation variations. The x-axis and y-axis in all the nine Scenarios represent the time and cache size, respectively.

Processor	4 cores, 4-way fetch and issue in-order for each, single threaded
Clock Frequency	2GHz
L1 D-Caches	Direct mapped, 32 KB, 64-byte block size, 3 cycle latency
L1 I-Caches	Direct mapped, 32 KB, 64-byte block size, 3 cycle latency
L2 Cache	8-way set associative, 4MB, 64-byte block size, 15 cycle latency
Memory	4GB, 200 cycle off-chip latency

Table 4.1. Platform setup.

Test cases and benchmarks. We evaluated the effectiveness of our reactive tiling approach with two classes of caches allocations. In the first class (synthetic allocations), cache allocations are generated by enforcing a partitioning policy on the shared cache. In the second class (SPEC2006 benchmarks), the cache allocations are determined by continuously monitoring the co-running application performance. In our implementation, we used a variant of utility-based cache partitioning scheme [35], though our approach can be easily integrated with other hardware/OS partitioning schemes as well. In the experiments we use only one reactive application, Symmetric Rank K Update Kernel (DSYRK) [1].

• *Synthetic Scenarios.* The rationale behind generating synthetic scenarios (see Figure 7) is to generate different cache allocation patterns. Broadly speaking, the allocations are monotonically increasing (#1, #4), monotonically decreasing (#2, #5), fluctuating between two values (#3), monotonically increasing and then decreasing (#7), monotonically decreasing and then increasing (#8), and finally constant (#6, #9). Even though other synthetic configurations could be generated, we believe that these configurations are representative of a large class of cache allocation scenarios.

• *SPEC2006 Scenarios.* The SPEC2006 scenarios are designed to test our approach under a realistic cache partitioning scheme, specifically, we used a variant of utility-based cache parti-

[1] DSYRK is a representative kernel of the family of level-3 Basic Linear Algebra Subprograms (BLAS) routines, these routines are heavily used in high performance computing and Linear Algebra Solvers

tioning scheme [35], to test our framework. In this setup, the cache allotment is performed by continuously measuring the IPC (Instructions Per Cycle) of each workload. Decisions for cache allocation, in this framework, are based on maximizing an objective function of some global metric such as the weighted speedup. In these experiments, we execute one reactive application with three SPEC2006 applications on the same multicore machine. We use a cache partitioner that dynamically (continuously) divides the shared L2 cache space across the applications (1 reactive and 3 SPEC2006) in the workload, and the reactive application modulates its tile sizes based on its cache allocations.

System Specification. We built our complete system in a full system simulator [4]. The challenging part in the implementation was to establish communication between the OS and the reactive application. Our implementation uses a general purpose register (eax) to achieve this communication. Using in-line assembly instructions, the size of cache allotment is communicated to the application through this register (the register holds the memory reference where the size of cache allotment is written). Cache allocations are written to the shared memory using SIM_Write_Phys_Memory($\cdots$) function call. The application reads the contents of this memory location (at safe points) to identify the best tile size. It then switches to appropriate tile size based on the current cache allotment. In our experiments, the applications are bound to cores using the sched_setaffinity($\cdots$) system call.

Scenario#	Static Tiles (seconds)	Reactive Tiling (seconds)	percent improvement
1	295.81	276.34	6.5 %
2	504.18	308.75	38.7 %
3	297.77	282.66	5.0 %
4	449.44	298.39	33.6%
5	375.42	306.30	18.4%
6	274.83	274.91	0.0%
7	304.63	282.52	7.2%
8	390.34	292.16	25.1%
9	312.26	313.01	-0.2%

Figure 4.2. Results from the synthetic allocation cache allocations

4.2 Experimental Results

Figures 8(a),(b),(c),(d) show the execution time (in seconds) for various versions of the tiled code for four reactive applications. Figure 8(a) shows profiled information for all possible tile sizes and cache allocations, for Symmetric Rank k Update Kernel. In this figure, X-axis represents various tile sizes, and Y-axis represents the performance (execution time) of each tile size. The vertical group of bars of a given tile size correspond to the execution time with varying cache allocations. Each group of vertical bars from left to right show cache allocations ranging from 32KB to 2048KB. In a similar manner, Figures 8(b), 8(c) and 8(d) give results for Matrix Multiplication,

Mix	SPEC2006 Mix	Static Optimal Tiles (sec)	Reactive Tiling (sec)	Percentage improvement	Workload IPC improvement
1	dsyrk, leslie3d, sjeng, specrand	257.93	203.02	21.2 %	0.6%
2	dsyrk, mcf, dealII, gcc	250.19	202.97	18.8 %	2.3%
3	dsyrk, lbm, GemsFDTD, calculix	326.02	285.13	12.5 %	5.4%
4	dsyrk, lbm, bzip2, gromacs	331.93	274.66	17.2 %	6.7%
5	dsyrk, hmmer, sjeng, omnetpp	335.11	290.82	13.2 %	13.2%
6	dsyrk, h264ref, gobmk, Xalan	319.88	268.77	15.9 %	0.2%
7	dsyrk, leslie3d, Xalan, sjeng	232.26	229.01	1.4 %	2.4%
8	dsyrk, astar, bzip2, calculix	301.56	244.90	18.7%	3.9%
9	dsyrk, astar, h264ref, bwaves	302.82	263.49	12.9%	-0.3%
10	dsyrk, bzip2, GemsFDTD, gromacs	293.01	254.92	12.9 %	-0.2%
11	dsyrk, calculix, cactusADM, hmmer	328.14	278.07	15.2 %	3.2%
12	dsyrk, cactusADM, dealII, astar	301.88	269.78	10.6 %	-2.4%
13	dsyrk, cactusADM, gromacs, lbm	339.06	285.63	15.7 %	2.1%
14	dsyrk, gobmk, mcf, gromacs	287.97	249.12	13.4 %	-0.4%
15	dsyrk, gcc, mcf, h264ref	306.58	290.80	5.14 %	0.0%

Figure 4.3. Results from utility based cache allocations

Symmetric Matrix Multiplication and In-Place Triangular Matrix Multiplication, respectively.

In the rest of our experiments, we use only one reactive application (dsyrk). The left table in Figure 9, gives the execution results with the best static tile size (for that application) and reactive tiling. One can see from the results that our reactive tiling strategy generates about 19.25% improvement over the best static tile size, when averaged over all synthetic scenarios shown in Figure 7. Note that each bar corresponds to the execution of an application with a fixed cache allocation and a fixed (static) tile size. The takeaway message from these results is that, depending on the available cache capacity, each application may prefer a different tile size.

The right table in Figure 9, on the other hand shows the percentage improvements reactive tiling brings over then best static tile i.e., being executed with SPEC2006 applications. We see that, on average our approach improves execution time by 10.52%.

Chapter 5

Related Work

In the chip-on-multiprocessor (CMP) domain, a large volume of literature investigated into various cache partitioning schemes [2, 5, 6, 7, 8, 13, 14, 15]. Typically, cache partitioning schemes can be broadly classified into 3 steps: Measuring, Partitioning and Enforcement of the partitioning policy in repetitive manner [2]. (1) Measurement: In terms of miss rate (or other metrics such as IPC), measure the performance of each application; (2) Partitioning: According to the performance measurement, find the optimized cache partitioning scheme by some objective functions; (3) Enforcement: Enforce the proposed partitioning scheme.

Stone et al. [5] studied optimal allocation of cache memory between two competing processes which minimizes the overall miss-rate of a cache. The focus of their work is miss rate as a function of cache allocation of individual competing processes. It shows that the optimal allocation occurs at a point where the miss-rate derivatives of the competing processes are equal. Suh et al. [6] proposed a dynamic cache partitioning method for simultaneous multithreading systems which can be applied to set associative caches at any partition granularity. It minimizes the overall cache miss rate wherein a cache miss will only allocate a new cache block to a thread if its current allocation is below its limit. Chang and Sohi [2] presented Cooperative Cache Partitioning (CCP) to allocate cache resources among threads concurrently running on CMPs. Unlike cache partitioning schemes that use a single partition repeatedly throughout a stable program phase, CCP adapts multiple time-sharing partitions to resolve cache contention.

Hsu et al. [7] investigated various partitioning metrics and found that simple policies like LRU replacement and static uniform partitioning cannot provide near-optimal performance. Kirk [8] developed a strategy that statically partitions the cache for each task in the task set to allow maximum cache performance. Suh et al. [13] presented an analytical cache model, which has been used to dynamically partition the cache, to accurately estimate the overall miss-rate for multiple applications running on CMP machine. Dybdahl and Stenstrom [14] proposed an adaptive shared/private partitioning scheme to exploit private cache locality and avoid inter-thread interference. Kim et al. [15] evaluated five cache fairness metrics that measure the degree

of fairness in cache sharing. By using these metrics, they proposes static and dynamic L2 cache partitioning schemes to optimize fairness in order to improve the system performance.

Compiler directed loop transformations [29, 30, 31, 32, 27, 24] such as tiling has been extensively studied in recent decades. A large volume of work has addressed the problem of selecting tile sizes which improve application performance [16, 20, 17, 18, 23, 19]. Coleman and McKinley [16] proposed a tile size selecting scheme by taking cache capacity and cache line size into account. Nikolopoulos [39] proposed a dynamic tiling scheme which swtiches between two tile sizes to prevent cache conflicts using copy and block layout mechanisms. Our approach differs from [39] in that it prevents maintenance of additional buffers and copying to these buffers. Further, our approach does not need to compute the linearised expression for block layout which could be expensive. Esseghir [20] presented an algorithm chooses the maximum number of complete columns that fit in the cache. Sarkar and Megiddo [17] introduced an analytical model to estimate the memory cost of a loop nest and an iterative search algorithm to find optimal tile sizes. Hartono et al. [18] considered tile sizes as parameter rather than a constant for imperfectly-nested loops so that dynamic optimizations can be implemented. Lam et al. [23] described a model for evaluating cache interference which evaluates reuse for one variable, and quantifies self-interference misses as a function of tile size. Zhao et al. [19] proposed a runtime optimization strategy to empirically search for ideal tile sizes.

Our reactive tiling scheme differs from the rest of the earlier proposed schemes as we consider the runtime effects of allocated shared cache space to the tile size. There are several advantages in our framework. First, the application can adapt itself according to the current system cache allocation. Second, a unified adaptive code is generated which can be used for all possible system cache configurations. This one time compilation strategy generates binaries for various target system configurations. The same executable can be ported to various systems with varying memory hierarchies. Third, when an application begins execution, the state of computation is preserved at each safe point. This is particularly important in application scenarios where the initial selection of parameters is done based on the current system configuration, the parameters become invalid as soon as the system configuration changes. Our method ensures that the application always picks the right set of program parameters even while the application is running. Last but not least, it provides a natural check pointing mechanism where the state of computation need not be discarded in search of optimal parameters.

Chapter 6

Concluding Remarks

The main contribution of this thesis is a reactive tiling strategy using which an application can react to OS/hardware based resource allocation decisions. In this strategy, at suitable execution points, the (reactive) application checks the amount of cache space (from a shared cache storage) made available to it (e.g., by the OS or by an hardware-based resource partitioner) and switches to the best tile size that goes with the new cache allocation. We tested the success of this strategy using synthetic allocations (that enforce pre-specified fix allocation patterns) as well as allocations coming from a utility based cache partitioner. Our experimental results reveal that the proposed reactive tiling approach improves over the best static tiles by 19.2% (on average) when using static allocations and 13.3% (on average) when using the allocations from the utility based cache partitioning.

Bibliography

[1] R. Bitirgen, E. Ipek and J. Martinez Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of MICRO*, 2008.

[2] J. Chang and G.S.Sohi Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of ICS*, 2007.

[3] S. Eyerman et. al, System Level Performance Metrics for multiprogramm Workloads, S. Eyerman, IEEE MICRO, 2008.

[4] P. S. Magnusson et al. Simics: a full system simulation platform. *Computer*, 35(2):50–58, 2002.

[5] H. S. Stone, J. Turek, and J. L. Wolf. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.*, 41(9):1054–1068, 1992.

[6] G.E. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *Proceedings of PDCS*, 2001.

[7] L. Hsu, S. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *proceedings of PACT*, 2006.

[8] D.B.Kirk. Process Dependent Static Cache Partitioning for Real-time Systems. In *proceedings of RTSS*, 1988.

[9] "Single-chip cloud computer," http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[10] "Teraflops research chip," http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[11] "IBM Power7 - smarter systems for a smarter planet" http://www.ibm.com/.

[12] "Amd many-cores processors," http://www.amd.com/.

[13] G.E. Suh, L. Rudolph, and S. Devadas. Analytical Cache Models with Applications to Cache Partitioning. In *proceedings of ICS*, 2001.

[14] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *proceedings of HPCA*, 2007.

[15] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *proceedings of PACT*, 2004.

[16] S. Coleman and K. S. McKinley. Tile Size Selection using Cache Organization and Data layout. In *proceedings of PLDI*, 1995.

[17] V. Sarkar and N. Megiddo. An Analytical Model for Loop Tiling and Its Solution. In *proceedings of ISPASS*, 2000.

[18] A. Hartono et al. Parametric Multi-level Tiling of Imperfectly Nested Loops. In *proceedings of ICS*, 2009.

[19] J. Zhao et al. Adaptive Loop Tiling for a Multi-cluster CMP. In *proceedings of ICA3PP*, 2008.

[20] K. Esseghir. *Improving Data Locality for Caches*. Master Thesis, Rice University, Houston, TX, USA, 1993.

[21] S. W. Williams. *Auto-tuning Performance on Multicore Computers*. Ph.D Thesis, University of California, Berkeley, CA, USA, 2008.

[22] B. C. Lee et al. Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-vector Multiply. In *proceedings of ICPP*, 2004.

[23] M. E. Lam, E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *proceedings of ASPLOS*, 1991.

[24] M. E. Wolfe. More Iteration Space Tiling. In *proceedings of SC*, 1989.

[25] F. Quillere, S. V. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, 2000.

[26] C.Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *proceedings of PACT*, 2004.

[27] U. Bondhugula et al. A Practical and Automatic Polyhedral Program Optimization System. In *proceedings of PLDI*, 2008.

[28] The Chunky Loop Generator. *http://www.cloog.org*.

[29] P. Feautrier. Some Efficient Solutions to the Affine Scheduling problem: I. one-dimensional time. *Intl. J. of Parallel Programming*, 21(5):313–348, 1992.

[30] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *proceedings of PPOPP*, 1995.

[31] M. Kandemir et al. Improving Locality Using Loop and Data Transformations in an Integrated Framework. In *proceedings of Micro*, 1998.

[32] A. W. Lim, G. I. Cheong, and M. S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *proceedings of ICS*, 1999.

[33] M. E. Wolfe and M. S. Lam. A Data Locality Optimizing Algorithm. In *proceedings of PLDI*, 1991.

[34] Iyer, R. CQoS: a framework for enabling QoS in shared caches of CMP platforms, *ICS*, 2004.

[35] Qureshi, M. K. and Patt, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *MICRO*, 2006

[36] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. In *proceedings of SC*, 1991.

[37] F. Irigoin and R. Triolet. Supernode Partitioning. In *proceedings of POPL*, 1988.

[38] Ancourt, Corinne and Irigoin, François Scanning Polyhedra with DO Loops. In *Proceedings of PPOPP*, 1991.

[39] Dimitrios S. Nikolopoulos. Dynamic tiling for effective use of shared caches on multithreaded processors. In *International Journal of High Performance Computing and Networking*, 2004.

[40] Lakshminarayanan et al. Parameterized Tiled Loops for Free In *proceedings of PLDI*, 2007.