Predicting the Impact of Optimizations for Embedded Systems

Min Zhao Bruce Childers Mary Lou Soffa
Department of Computer Science
University of Pittsburgh, Pittsburgh, 15260
{lilyzhao, childers, soffa@cs.pitt.edu}

ABSTRACT
When applying optimizations, a number of decisions are made using fixed strategies, such as always applying an optimization if it is applicable, applying optimizations in a fixed order and assuming a fixed configuration for optimizations such as tile size and loop unrolling factor. While it is widely recognized that these fixed strategies may not be the most appropriate for producing high quality code, especially for embedded systems, there are no general and automatic strategies that do otherwise. In this paper, we present a framework that enables these decisions to be made based on predicting the impact of an optimization, taking into account resources and code context. The framework consists of optimization models, code models and resource models, which are integrated for predicting the impact of applying optimizations. Because data cache performance is important to embedded codes, we focus on cache performance and present an instance of the framework for cache performance in this paper. Since most opportunities for cache improvement come from loop optimizations, we describe code, optimization and cache models tailored to predict the impact of applying loop optimizations for data locality. Experimentally we demonstrate the need to selectively apply optimizations and show the performance benefit of our framework in predicting when to apply an optimization. We also show that our framework can be used to choose the most beneficial optimization when a number of optimizations can be applied to a loop nest. And lastly, we show that we can use the framework to combine optimizations on a loop nest.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors– *Compiler; Optimization*

General Terms
Design, Experimentation, Performance

Keywords
Embedded Systems, Optimizing Compilers, Loop Optimizations, Prediction, Code Models, Resource Models, Optimization Models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES '03, June 11-13, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-647-1/03/0006…$5.00.

1. INTRODUCTION
Embedded systems have had tremendous growth in the last few years and account for more than 99% of microprocessors produced every year, with 500 million embedded microprocessors to be sold in 2005 [T99, AMP00]. Embedded systems have ever-increasing demands for functionality and performance, while requiring very low cost. Optimizing compilers can play a role in achieving cost/performance goals for embedded systems by applying code transformations to improve performance, minimize memory footprint, and lower energy consumption.

While traditional compiler optimizations have been applied to improve performance, current compiler techniques do not always achieve the best potential performance. For cost-sensitive embedded systems, it is paramount to achieve the best possible performance, given available resources. For example, with aggressive optimization, it may be possible to reduce processor clock rate after applying code transformations while still meeting performance goals (which may have other system effects, such as reducing energy consumption). However, current optimization strategies do not always achieve the performance goals. Indeed, it is well known in the compiler community that optimizations may degrade performance in certain circumstances. The difficulty is that current techniques cannot always determine when it is beneficial or harmful to apply an optimization.

Another problem for performance is the combination of optimizations, which in some cases, leads to better performance than the individual application of optimizations [SR92, CC95]. We do not currently have a systematic way of determining if and which combinations of optimizations are helpful. Also when there is more than one optimization that is applicable, one of them may be more valuable to apply than the others. Ideally, we would like to select the one that has the biggest performance impact. Moreover, because of enabling and disabling interactions, the order of applying optimizations from a suite of optimizations can have an impact on performance [ZCW02, WS97, TVVA03]. Typically, the compiler designer decides on an optimization order using her experience and just applies optimizations in that order. Lastly, the configuration of a particular optimization can impact performance (e.g., how many times to unroll a loop, tile size, etc.) [MCT96]. In all of these cases, although we have techniques for handling some of these problems in isolation, there is no general, uniform way to effectively address the problems.

Ideally, we would like to be able to predict the performance impact of optimizations before applying them to evaluate their efficacy. Prediction is difficult because the impact of a given optimization varies markedly and is determined by a number of factors, including the target machine architecture (e.g., cache

configuration and the number of available registers), the optimization configuration, and the code context where the optimization is applied. What is needed is a uniform way of expressing the variations that is useful for predicting the impact of optimizations.

In this paper, we present a **F**ramework for **P**redicting the impact of **O**ptimizations (FPO) for some objective (e.g., performance, code size or energy). The framework consists of three types of models: optimization models, code models and resource models. The structure of our framework, as shown in Figure 1, includes: (1) optimization models that represent the characteristics of the optimizations in terms of how they will impact some objective, both qualitatively and quantitatively, (2) resource models that parameterize the target machine configuration, and (3) code models that abstract information about the application code. By integrating the models, a "benefit" value is produced that represents the benefit of applying an optimization in a code context for the objective represented by the resources.

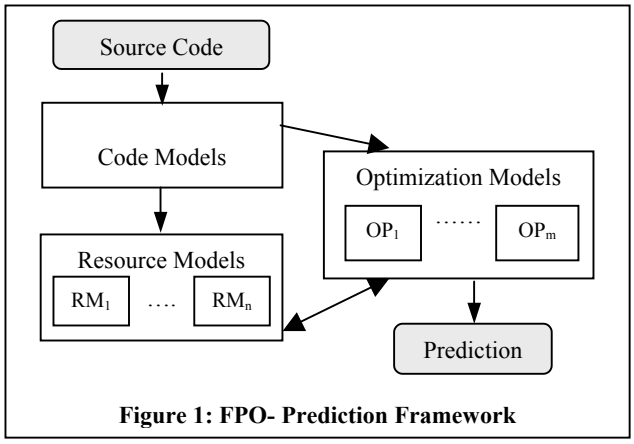

Figure 1: FPO- Prediction Framework

Based on predictions from our framework, the problems listed above can be tackled. Using particular optimization, resource, and code models, the framework can be used to predict whether it is beneficial or not to apply the optimization in a code context. The optimization models (different configurations or different optimizations) can be combined and thus we can predict the benefit of combining optimizations rather than applying them one at a time. When more than one optimization can be applied in a code segment, the framework can be used to predict the best one to apply. Lastly, the prediction value can be used as an objective function when using search techniques such as genetic algorithms and AI planning to find the best configuration or the best order to apply optimizations.

Since many factors impact the overall performance of a program, including cache performance, register allocation and instruction scheduling, it is very difficult to analytically predict the impact of optimizations on performance [CST01]. Our approach is to isolate each of these factors and predict the impact using one factor at a time with the ultimate goal of integrating all the factors. In this paper, we focus on using the framework to predict the impact on cache performance for embedded processors.

As the disparity between processor and main memory speed increases by approximately 50 percent per year, the use of caches with high hit rates has become critical for performance [GMM99]. Data caches are designed to exploit locality, and naturally they work best for programs that have high locality. Some optimizations are designed to improve cache performance by rearranging code to have better locality. However, other optimizations are not designed specifically for this purpose and may negatively impact cache performance and the overall performance. We develop an instance of FPO, FPO-cache, to predict the impact of applying an optimization on cache. Since loop behavior dominates cache performance [MT96], we focus on loop optimizations: Our code and optimization models represent the characteristics of the loops and optimizations that impact cache performance. We also use a model of cache behavior for the array referencing pattern that estimates the cache cost of executing a code segment. After determining the impact of a loop optimization on cache performance using FPO-cache, the code optimizer can decide whether it is beneficial to apply the optimization, or given a set of valid optimizations, decide which one should be applied for the best cache performance.

In the next section, we provide an overview of FPO-cache and describe our technique to predict the impact of optimizations on cache performance, including our code model, optimization models and cache model. Experimental results are reported in Section 3. We evaluate FPO-cache for predicting the cache performance impact of applying loop optimizations. We present experimental results that indicate that always applying an optimization can, in fact, degrade performance. We describe the performance benefit of selectively applying an optimization only when a prediction indicates an improvement. Also we experimentally show the prediction accuracy of FPO-cache. Finally, we demonstrate the usefulness of FPO-cache for choosing the most beneficial optimizations and combining optimizations. Related work is briefly given in Section 4, followed by conclusions and future work in Section 5.

The contributions of this paper include:

- A unique framework that integrates optimization, resource and code models to predict the impact of optimizations without applying them,
- An instance of that framework for predicting the impact of loop optimizations on cache,
- Analytical optimization models that capture the characteristics of the optimizations as to how they impact cache performance,
- Experimental results showing that always applying optimizations does degrade cache performance in some cases and that better performance can be achieved by using our framework to selectively apply optimizations when there is a predicted benefit, and
- Demonstration of the usefulness of the framework in combining optimizations and selecting the best optimization to apply for embedded systems.

2. FPO-cache

To predict the impact of optimizations on cache locality, we first extract information about a loop nest and represent it with a code model. We consider loop nests because they dominate cache performance for a given code [MT96]. We assume the array is arranged in row order without loss of generality. Next, we use optimization models to express the characteristics that affect cache performance for an individual optimization. We then use a

cache model to estimate the number of cache misses of executing a loop ("cache cost"). By integrating these models, we can predict the impact of applying loop optimizations. In the next sections, we present these models, starting with our code model.

2.1. Code Model

To predict the impact of optimizations on cache, we need to express those code characteristics that affect the cache, which are a loop's header and the sequence of array references in a loop body.

DEF 1 A ***loop header***, $\oint_{lb}^{ub} step$, consists of a lower bound (*lb*), upper bound (*ub*), and iteration step (*step*).

DEF 2 A ***reference*** is a static read or write in a program, while a particular execution of that read or write at run time is a memory access [GMM99].

DEF 3 An ***array reference*** is a reference that refers to an array element and includes the name of the array and its access function (subscript). Because optimizations usually change the access functions (and not the name of the array), we use an equation, $\overrightarrow{Ref} = A \times \vec{I} + \vec{C}$, to represent the access function of an array reference. Here, A is the access matrix, $\vec{I}$ is the loop index vector and $\vec{C}$ is the constant vector [HKVI02]. This equation can be written as:

$$\begin{bmatrix} sub0 \\ \vdots \\ subd-1 \end{bmatrix} = \begin{bmatrix} A00 & \cdots & A0(N-1) \\ \vdots & \ddots & \vdots \\ A(d-1)0 & \cdots & A(d-1)(N-1) \end{bmatrix} \times \begin{bmatrix} I0 \\ \vdots \\ IN\text{-}1 \end{bmatrix} + \begin{bmatrix} C0 \\ \vdots \\ Cd-1 \end{bmatrix}$$

In our model, we use the access matrix (A) and constant vector (C) to represent an array reference. The loop index variables, $\vec{I}$, are represented by the loop header.

DEF 4 An ***array reference sequence***, $\langle R \rangle$, consists of all array references in a loop body in the order that they appear textually in the code.

DEF 5 A ***loop***, $\oint_{lb}^{ub} step \langle R \rangle$, consists of its header and array reference sequence.

DEF 6 A ***perfect loop nest***, $\oint_{N-1\ lb}^{ub\ step} \cdots \oint_{1\ lb}^{ub\ step} \oint_{0\ lb}^{ub\ step} \langle R \rangle$, is a sequence of loops enclosing the same array reference sequence. For convenience, we put a number under the loop header to express its order in the loop nest. Although all of the loop nests that we consider in this paper are perfect loop nests, our technique can be extended to handle non-perfect nested loops by including the loop index $\vec{I}$ in every array reference.

DEF 7 A ***loop nest sequence,*** $\langle LN \rangle = \langle ln1, ln2, \ldots \rangle$, is a sequence of loop nests in the order they appear in the code. A loop nest sequence that has one loop nest is the loop nest itself.

```
for ( i=0; i<N; i++)
    for ( j=0; j<N; j++)
        a[i] = a[i] + b[j][i]*c[i][j]+c[i+1][j];
```

(a) Original loop nest

$$\oint_{1\ 0}^{N-1\ 1} \oint_{0\ 0}^{N-1\ 1} \langle\langle (Aa, Ca), (Ab, Cb), (Ac1, Cc1), (Ac2, Cc2), (Aa, Ca) \rangle\rangle$$

(b) Code model for the loop nest

Figure 2. A loop nest and its code model

Figure 2 shows an example of a loop nest and its code model, where (Aa, Ca) represent the access matrix and constant vector of the array reference a[i] (same for the array references b and c.)

2.2. Optimization Models

As was said, our optimization models capture the characteristics that affect cache, which include loop headers and array references. Loop headers give the total number of memory accesses for an array reference. The loop organization and array reference pattern determine how the memory accesses are ordered. Different orders result in different data reuse and thus different amounts of cache misses. Because an optimization affects the loop headers and array references structure, we use a function to describe the impact of an optimization.

DEF 8 ***Impact function*** of an optimization, $f_{opt}(\langle LN \rangle) = \langle LN' \rangle$, is a function that maps an original loop nest sequence to a new loop nest sequence.

We develop an impact function for every loop optimization considered in this paper. In the following sections, we present our optimization models, including the impact functions, for loop interchange, unrolling, tiling, reversal, fusion, and distribution [BGS94].

2.2.1. Loop Interchange

Loop interchange exchanges the position of two loops in a loop nest. The optimization model for loop interchange is illustrated in Figure 3. The impact function, $f_{interchange}$, maps an original loop nest to a new loop nest, according to the semantics of loop interchange. Essentially this function exchanges *lb, ub* and *step* of loop *i* with that of loop *j*. It also changes the array reference sequence $\langle R \rangle$ by a function $g\langle R \rangle$. This function determines the new array reference sequence for the transformed loop by applying *h(r)* on every reference *r* in $\langle R \rangle$. Function *h(r)* computes a new array reference by exchanging column *i* and *j* in the access matrix *A* from *r*'s reference equation. *l(A)* handles the column interchange. The constant vector (C) for *r* is unchanged.

Consider the example in Figure 2. Using the model in Figure 3, we determine the new loop nest. The new header is determined by exchanging *lb*, *ub*, and *step* for loop l_i and l_j. The new array reference sequence, $\langle R' \rangle = \langle r0', r1', r2', \ldots, r4' \rangle$, is determined by changing the access matrix of every array reference in $\langle R \rangle$. For

example, the access matrix of a[i] is changed from $\begin{bmatrix}1 & 0\end{bmatrix}$ to $\begin{bmatrix}0 & 1\end{bmatrix}$ and b[j][i] is changed from $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$ to $\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$.

INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ and interchange is legal for loops *i, j*;

$$f_{Interchange}(\oint_{N-1}\cdots\oint_{i}\cdots\oint_{j}\cdots\oint_{0}\langle R\rangle)=\oint_{N-1}\cdots\oint_{j}\cdots\oint_{i}\cdots\oint_{0}g(\langle R\rangle)$$

where $g(\langle R\rangle)=\langle\forall(r\in\langle R\rangle)\,h(r)\rangle$,

$h(r)=(l(A),C)$, and $l(A)=A[:][i]\leftrightarrow A[:][j]$

Figure 3: Loop Interchange Model

2.2.2. *Loop Unrolling*

Loop unrolling duplicates a loop's body a number of times [BGS94]. It is commonly understood that loop unrolling has little impact on cache performance when register pressure is not considered. However, we modeled the loop unrolling to demonstrate the performance of our models. The optimization model for loop unrolling is shown in Figure 4.

INPUT: $\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle$ and unroll factor U;

$$f_{unroll}(\oint_{N-1}\cdots\oint_{1}\oint_{0}\langle R\rangle)=\langle ln_{unroll},ln_{rest}\rangle \text{ where}$$

$$ln_{unroll}=\oint_{N-1}\cdots\oint_{1}\oint_{0}^{step\times U}g(\langle R\rangle) \text{ and}$$

$$ln_{rest}=\oint_{N-1}\cdots\oint_{1}\oint_{0}^{\left\lceil\frac{ub+1}{U}\right\rceil\times U}\langle R\rangle$$

$$g(\langle R\rangle)=\langle R\rangle\,\hat{}\,(\bigwedge_{i=1}^{U-1}\langle\forall(r\in\langle R\rangle)\,h(r,i)\rangle) \text{ and}$$

$$h(r,i)=(A,l(C,i)) \text{ and}$$

$$l(C,i)=\forall(s\in\{a\mid A[a][N-1]\neq 0\})\,C[s]+i$$

Figure 4: Loop Unrolling Model

The impact function $f_{unrollling}$ maps an original loop nest to two nested loops (one for the unrolled loop and one for the possible leftover iterations) according to the semantics of loop unrolling. In the unrolled loop nest, the *step* of the innermost loop is changed to $step\times U$ (U is the unroll factor) and the array reference sequence, $\langle R\rangle$, is changed by a function *g*, which combines $\langle R\rangle,\langle R1\rangle,\langle R2\rangle,\cdots,\langle RU-1\rangle$ together. A reference $\langle Ri\rangle$ is determined by applying the function $h(r,i)$ on every array reference, *r*, in $\langle R\rangle$. Function $h(r,i)$ models how the access matrix and constant vector of a reference are changed. It keeps the access matrix unchanged and applies *l(C,i)* on the constant vector. Essentially, *l(C,i)* changes *C* by adding *i* to those dimensions that have the innermost loop control variable. In the loop nest for the leftover iterations, the *lb* of the innermost loop is changed to $\left\lceil\frac{ub+1}{U}\right\rceil\times U$ and the array reference sequence, $\langle R\rangle$, is unchanged.

Use the example from Figure 2 to illustrate our model, supposing that the unroll factor is two. With the model from Figure 4, the unrolled loop's header becomes, $\oint_{1}^{N-1}{}_{0}\oint_{0}^{N-1}{}_{0}$, from the rolled loop's header, $\oint_{1}^{N-1}{}_{0}\oint_{0}^{N-1}{}_{0}$, by doubling the *step* of the innermost loop. The array reference sequence for the unrolled loop is $\langle r0,r1,\cdots,r5,\cdots,r9\rangle$, where *r5* to *r9* is determined by keeping the access matrix and changing the constant vector of r0 to r4 in $\langle R\rangle$. For example, *r6* (b[j+1][i]) has the same access matrix $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$ as $r1$ (b[j][i]), but a different constant vector $\begin{bmatrix}1\\ 0\end{bmatrix}$. Second, we determine the loop nest for the leftover iterations. Its loop header is $\oint_{1}^{N-1}{}_{0}\oint_{0}^{\left\lceil\frac{N}{2}\right\rceil\times 2}{}_{0}$ and its array reference sequence is unchanged.

2.2.3. *Loop Tiling*

Loop tiling improves cache reuse by dividing an iteration space into tiles and transforming the loop nest to iterate over them [BGS94]. The optimization model for loop tiling is shown in Figure 5.

The impact function, f_{tiling}, maps an original loop nest to a new loop nest by changing its loop header by function *g* and changing its array reference sequence $\langle R\rangle$ by function *f*. Essentially, function *g* adds $\oint_{N+n-1}^{ubn}{}_{lbn}^{tsn}\cdots\oint_{N}^{ub1}{}_{lb1}^{ts1}$ to the outermost and changes *lb* and *ub* of loops to be tiled. (The input to the model specifies the number of loops to be tiled, *n*, their index in the header sequence $t1,t2,\cdots,tn$ and their tile size, $ts1,ts2,\cdots,tsn$.) The *lb* of l_{ti} changes to the control variable of l_{N+ti-1} (represented as x_i). The *ub* of l_{ti} changes to a function *h(i)*, which gets the minimum number of original *ub* and $(xi+tsi-1)$. On the other hand, function $f(\langle R\rangle)$ changes the access matrix (A) by function *l(A)* of every array reference in $\langle R\rangle$, where function *l(A)* adds *n* columns of zero to A's first *n* columns. The constant vector (C) does not change.

For the example in Figure 2, if we tile l_i and l_j with tile size 64, using the model shown in Figure 5, we get the new loop header as $\oint_{3}^{N-1}{}_{0}^{64}\oint_{2}^{N-1}{}_{0}^{64}\oint_{1}^{\min(N-1,x2+63)}{}_{x2}^{1}\oint_{0}^{\min(N-1,x1+63)}{}_{x1}^{1}$. The access matrix of

every array reference is changed, e.g., b[j][i] is changed from $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ to $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

INPUT: $\oint_{N-1} \cdots \oint_{1} \oint_{0} \langle R \rangle$, tiling loops $t1, \cdots, tn$, with tile size $ts1, \cdots, tsn$ respectively;

$$f_{tiling}(\oint_{N-1} \cdots \oint_{tn} \cdots \oint_{t1} \cdots \oint_{0} \langle R \rangle) = g(\oint_{N-1} \cdots \oint_{tn} \cdots \oint_{t1} \cdots \oint_{0}) \; f\langle R \rangle \text{ where}$$

$$g(\oint_{N-1} \cdots \oint_{tn} \cdots \oint_{t1} \cdots \oint_{0}) = \oint_{N+n-1}^{ubn} \cdots \oint_{N}^{ub1} \oint_{N-1}^{h(n)} \cdots \oint_{t1}^{h(1)} \cdots \oint_{0}$$

$$h(i) = \min(ubi, xi + tsi - 1) \text{ and}$$

$$f\langle R \rangle = \langle \forall (r \in \langle R \rangle) \; (l(A), C) \rangle \text{ where } l(A) = [0A]$$

Figure 5: Loop Tiling Model

2.2.4. Other Loop Optimizations

We also develop impact function for loop reversal, loop fusion and loop distribution. The detailed optimization models for these loop optimizations are described in our technical report at: http://www.cs.pitt.edu/copa/FPO.pdf

2.3. Cache Model

We use a cache model to estimate the cache cost of executing a loop nest. This model indicates how a given reference pattern affects cache misses (and hits) under the assumption of a single issue in-order pipelined processor with a blocking cache (see Section 3). To improve locality, we want to reduce the number of cache misses, and in evaluating the impact of an optimization, we want to know whether the number of cache misses is decreased by the optimization.

Because some array references may access the same cache line in the same or different iteration (due to group temporal or spatial reuse), we group references to avoid over estimating the number of cache misses when a reference may access a cache element that has been previously loaded. We adapt Mckinley et al.'s *RefGroup* algorithm [MCT96] to formulate *RefSet* using our code model representation to calculate group spatial and temporal reuse with respect to the innermost loop. We consider two references $r1$ (A_1, C_1) and $r2$ (A_2, C_2) that refer to the same array to belong to the same *RefSet* if:

(1) $A1 = A2$, $\forall ik$ (i_k is the row index of the none-zero elements in the last column of A_1) $|C1[ik] - C2[ik]| = p \times step_{N-1}$ (p is a positive integer and $p \le 2$, $step_{N-1}$ is the iteration step of the innermost loop), and all other i_p ($ip \neq ik$), $C1[ip] = C2[ip]$ or

(2) $A1 = A2$, $C1[i] = C2[i]$ ($0 \le i < d$-1), and $|C1[d-1] - C2[d-1]| < cls$ (cls is the cache line size, and d is the dimension of the array, as described in DEF3).

Condition 1 accounts for group temporal reuse, and condition 2 accounts for group spatial reuse.

Once we account for group reuse, we can calculate the cache misses of a representative array reference, say R_α, in a *RefSet*. Initially, we use McKinley et al.'s cache cost model. While their model accurately estimated cache misses under some circumstances, it did not have sufficient overall accuracy needed to achieve good results for all of our optimization models. The reason is that it handles cache conflict misses in a simple manner and did not accurately reflect all possible sources of conflict misses.

Cache conflicts are difficult to predict and estimate [TFJ94]. From our own experiments, we found that cache conflict misses can vary widely with slight variations in the problem input size. Ghosh et al. [GMM99] proposed a precise algorithm, Cache Miss Equation (CME), to generate a set of equations for cold and replacement misses. The solutions to these equations represent all compulsory and conflict misses. However, finding all reuse vectors and setting up complete cache miss equations is very complex. Instead, our goal was to develop a more feasible and practical model that tailors Ghosh's scheme to our specific problem of predicting the impact of loop optimizations on cache performance. We simplified Ghosh's model to calculate the cache misses of R_α. Suppose that *TI* is the total number of iterations in the loop nest, *cls* is the cache line size, and *FP* is the footprint R_α (i.e., how many different elements it access over all the iterations), *CRT* is the fraction of R_α's self temporal-reuse that cannot be realized (realizing a reuse means a reuse can result a cache hit) and *CRS* is the fraction of R_α's self spatial-reuse that cannot be realized. We estimate the cache misses of R_α to be:

$$CM(R\alpha) = TI \times (\frac{FP}{TI} \times (1 - CRT) + CRT) \times (\frac{1}{cls} \times (1 - CRS) + CRS) \quad (1)$$

We compute *CRS* and *CRT* in a way similar to the CME approach by solving a set of equations that sets the cache block address of R_α equal to that of other references within its reuse distance to find possible conflicts. The reuse distance is the number of iterations between a reuse and its previous access. For example, in Figure 2, b[j][i]'s spatial reuse distance is *N*, because an access in iteration (i, j) can be spatially reused by another access in iteration $(i+1, j)$, which is *N* iterations behind. With this approach, we take into account the cache conflicts in an accurate manner. We illustrate how to compute *CRS* and *CRT* for b[j][i] in Figure 2. Suppose that we have direct-mapped cache. First according to b[j][i]'s spatial reuse distance *N*, we set up a set of equations to get *CRS* for b[j][i], including:

$$\forall t \in [0, N-1] \;\; Addr(b[j][i]) = Addr(c[i][j+t]) \quad (2)$$

$$\forall t \in [0, N-1] \; Addr(b[j][i]) = Addr(c[i+1][j+t]) \quad (3)$$

$$\forall t \in [1, N] \quad Addr(b[j][i]) = Addr(b[j+t][i]) \quad (4)$$

$$Addr(b[j][i]) = Addr(a[i]) \quad (5)$$

The solutions to every equation represent all the iterations where b[j][i] conflicts with another reference. Because of the direct mapping, the total number of iterations that b[j][i] will be evicted by another reference will be the union of these solution sets. We compute *CRS* by dividing the total number of conflict iterations

by the total number of iterations. As b[j][i] has no temporal reuse, *CRT* equals one.

2.4. Integration of the Models

To integrate the code and optimization models with the cache model, we extract the loop nests from the original code and express them using our code model (described in Section 2.1). Then we input the code model and the optimization input parameters (shown in optimization models) into an optimization model and get a new code model that represents the optimized code. Finally we feed the original code model and the optimized code model into the cache model. With a cache configuration, the cache model estimates the cache misses according to the representation of the code model. We predict the impact of an optimization by determining the difference between the cache misses of the original and the optimized code models.

3. EXPERIMENTAL RESULTS

To evaluate the effectiveness of FPO-cache, we implemented our models and extensively tested them using several common and embedded benchmark loops from the PERFECT suite [BCK88], MediaBench [LPM97], DSPStone [ZMS94], and other researchers [HKVI02]. There are two types of benchmarks: those with a single loop nest (*alv, irkernel, lgsi, smsi, srsi, tfsi, tomcat3, biquad_N, lms, gdevcdj* and *pegwit*) and those with multiple loop nests (*adi, aps, eflux, tomcat, vpenta,* and *bmcm*). The benchmarks have from one to nine loop nests and from four to thirty two array references in a loop nest.

To experimentally evaluate our approach, we use the SimpleScalar *sim-outorder* microarchitecture simulator [BA97]. In our experiments, we use a 1KB direct-mapped data cache with 32-byte block size. Using a small cache with scaled working sets allows us to investigate the impact of different sized working sets without suffering the high simulation times required for large data sets. The performance numbers that we present will scale to other cache configurations and working set sizes.

In our performance evaluation, we model an embedded processor pipeline with in-order single issue and a critical-word first non-blocking cache. The processor has a two entry load-store queue and can sustain up to two cache misses before stalling. There are three reasons for this choice. First, in the embedded market, this model is similar to several popular processors, including MIPS' 4Kp (R4000), ARM's 94x series, and IBM's PowerPC 405. Second, although our cache model assumes a blocking cache and our performance evaluation is on a non-blocking cache (a more realistic assumption), we found that the non-blocking cache with a two entry load-store queue has similar performance to the blocking case for our array-based benchmarks. Third, we used this in-order single issue model to avoid other performance effects (e.g., hardware-based dynamic scheduling, speculative execution, and branch prediction), which may confuse our analysis of the results. Being able to model the impact of dynamic scheduling on cache performance is a separate issue that is beyond the scope of this work.

Using our benchmark loops, we investigated the benefit of our models in improving the application of loop optimizations. A tool was developed that takes a loop nest and, based on our models, predicts the impact of a loop optimization on cache performance. With our tool, we first investigated the impact of always applying an applicable optimization to motivate the need for our framework. Next, we show the importance of selectivity in applying an optimization and how it can improve performance over the "always applying" heuristic. We then validate our framework and demonstrate its accuracy in predicting the impact of an optimization. Finally, we describe two uses of our FPO-cache in selecting the most beneficial optimization and combining optimizations.

3.1. Always Applying an Optimization

Always applying an applicable optimization can lead to a performance degradation in some cases. Such a simple heuristic of “always applying” is not sufficient in making decisions about when to apply an optimization. Figure 6A shows how always applying an optimization can lead to significant performance penalties. This figure shows the percentage change in performance (i.e., cycle count) when always applying an optimization versus not applying the optimization. Several benchmarks were run with varying trip counts to explore the effect of different configurations of a loop on whether to apply an optimization or not. For the benchmarks where the configuration was varied, only two trip counts are shown. One trip count comes directly from the benchmark, while the other is at a point that has significant conflict cache misses. Although the results are not reported here, we varied the trip count for these benchmarks from 50 to 200 and the first case is near the average for all trip counts for a benchmark.

The figure demonstrates that across all benchmarks and optimizations that we considered, applying loop optimizations has significantly different performance impacts based on both a specific loop nest and the exact configuration of a loop nest. For example, loop interchange has a performance impact that varies from a 120% degradation to a 55% improvement. Also, for a specific configuration of a loop nest (i.e., different trip counts) the impact varies. In the case of interchange for the *lgsi* benchmark, there is a 4% performance degradation for a trip count of 98 and a 8.3% performance improvement for a trip count of 128. Although the figure does not show loop unrolling, distribution, or fusion, we used our models to predict their impact. First, as expected, loop unrolling had no benefit to data cache locality. Of course, it had other non-cache related benefits such as reducing the total number of branch tests, improving the scheduling window and changing register pressure. Second, distribution had a 17.8% degradation when applied to *alv* with a trip count of 100 and a 1.2% improvement when applied to *alv* with a trip count of 128. Finally, on *tomcat3*, fusion had a very small benefit (0.8%) for a trip count of 100 and a 2.8% degradation for a trip count of 128. Optimizations may improve the performance for one trip count while degrade the performance for another. This trend for the single loop nest benchmarks is also true for the complex benchmarks with multiple loop nests. Here, interchange has a performance range from a 2.5% degradation to a 55% improvement. Tiling shows a similar trend, with the *aps* having a 26.2% performance improvement and *vpenta* having a 1% performance degradation.

As this figure shows, the strategy of always applying an applicable loop optimization is a dangerous one that may indeed lead to significant performance degradations. Of course, in some

cases, this strategy works, but it is hard to know when it will work and

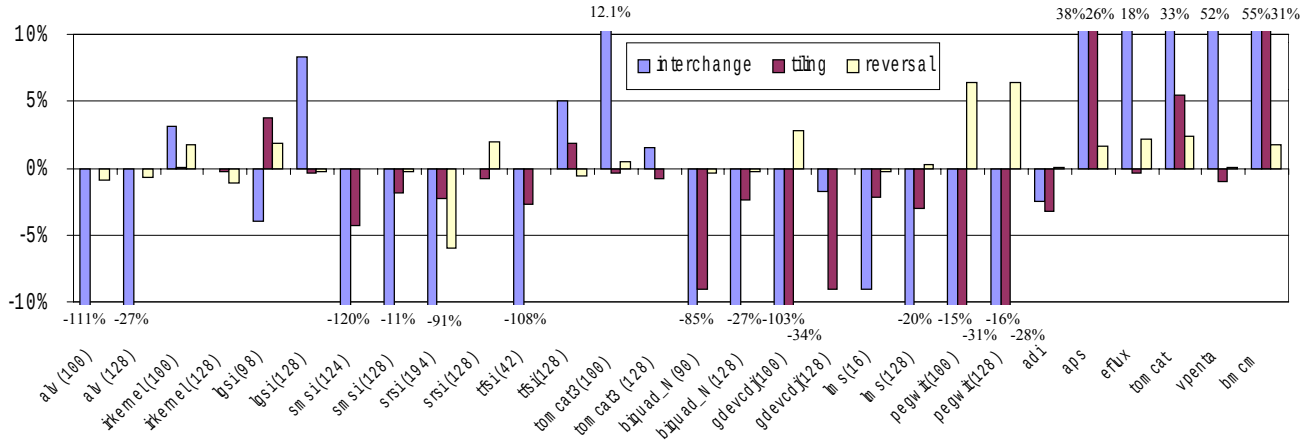

Figure 6A. Performance Impact of Always Applying an Optimization
(Trip counts are in the parentheses after the benchmark name.)

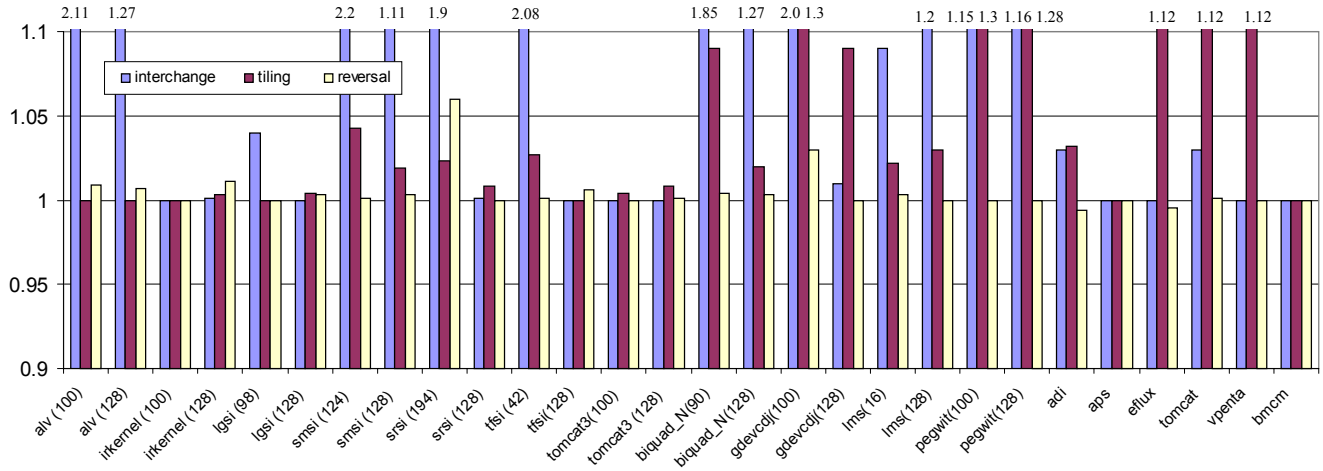

Figure 6B. Improvement of Selectively Applying vs. Always Applying

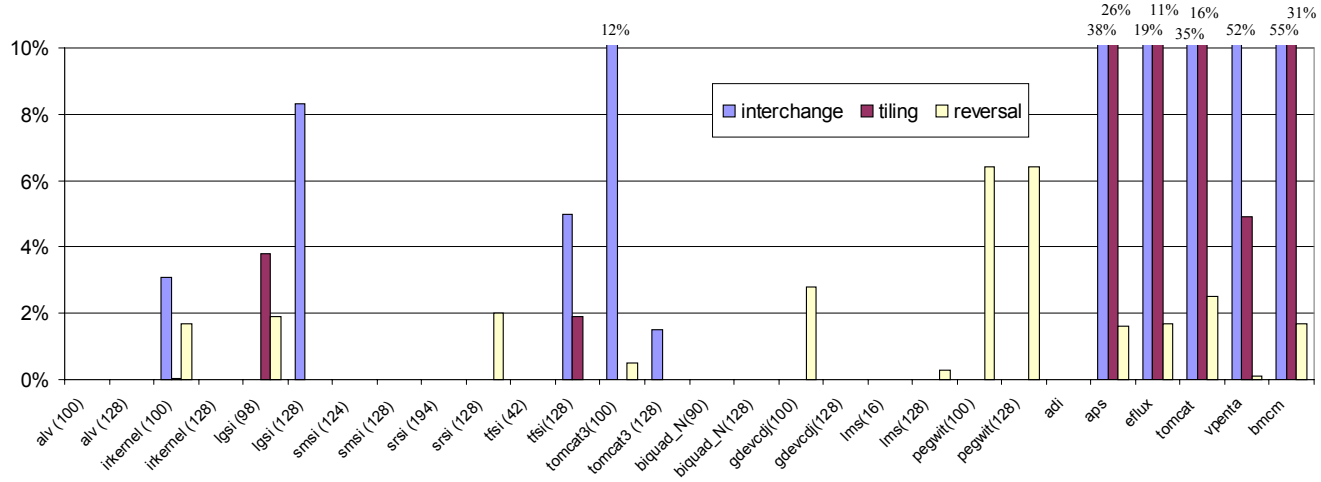

Figure 6C. Performance Impact of Selectively Applying an Optimization

when it will not. Instead of blindly applying an optimization, a more selective approach can be taken with our framework. It can be used to predict when to apply an applicable optimization without actually applying it.

3.2. Impact of Optimization Selectivity

By selectively applying an optimization, the cases where performance is degraded can be avoided, which can have a significant effect. Figure 6B shows the performance improvement of selectively applying an optimization over always applying it. The improvement is relative to always applying the optimization and demonstrates the effect of selectivity. For the single nest benchmarks, a performance improvement implies that an optimization was not applied. For example, the benchmark *alv* with a trip count of 100, selectively deciding not to apply loop interchange has twice the performance of applying it. When performance is not improved both always applying and selectively applying an optimization had the same effect.

For interchange on the single nest benchmarks, optimization selectivity has a performance improvement of 0 to 120%. The large improvements in this case are due to the large degradations from always applying interchange (see Figure 6A). Although loop tiling shows a slight improvement due to selectivity, it does not have as much an improvement as interchange because the degradation from always applying the optimization is less. Reversal is similar to the tiling case. Distribution and fusion also showed improvements when applied with selectivity. With selectivity, unrolling was not applied since it does not have any benefit to cache performance. For all single nest benchmarks and optimizations considered, a selective approach with our models never results in a performance degradation over always applying an optimization. Indeed, the model captures the points at which an optimization is harmful as well as the points at which an optimization is helpful.

The rightmost bars in the figure show the effect of selectivity on benchmarks with multiple loop nests. In these cases, interchange with selectivity has a small performance improvement for *adi* and *tomcat*. A similar trend is true for loop reversal. However, in the case of loop reversal, two points (*eflux* and *adi*) are shown where our model mispredicts the benefit of applying an optimization and results in a small performance degradation over always applying reversal. The situation is different for tiling where selectivity has a significant difference. For *eflux*, *tomcat*, and *vpenta*, there is a performance improvement of 1.12.

While Figure 6B shows the advantage of selectively applying an optimization, it does not show the actual improvement in execution time due to selectivity. Figure 6C shows how cycle count is improved. For the single nest benchmarks, performance is improved by deciding not to apply an optimization when it would be harmful and by applying an optimization when it would help. For the points where our model correctly decided not to apply the optimization, there is no reduction in actual cycle count. However, by selectively deciding not to apply interchange, the penalty of optimizations can be avoided.

In Figure 6C, the cases with multiple loop nests are very compelling with selectivity resulting in a cycle count improvement over always applying an optimization for some cases. Consider the *tomcat* benchmark and the tiling optimization. Tiling results in a 15.5% improvement in cycle count by selectively applying the optimization to some loop nests and not to others within the same program. In comparison, always applying tiling achieved only a 5.4% improvement in cycle count.

3.3. Model Accuracy

To use FPO-cache to select whether to apply an optimization or not, we must ensure that the models are useful in predicting the impact of an optimization on cache performance. To validate our models, we ran the original benchmarks and optimized ones with our simulation framework and compared the predictions from FPO-cache against the simulation results.

First we compared the predictions of cache miss reductions against the simulation results. When integrated with a simpler cache model [MCT96], FPO-cache could not make correct predictions in some cases, while with our cache model, FPO-cache predicted more accurately. Figure 7 shows an example how predictions compared with simulation for interchange on *irkernel* with varying trip counts. With a simple cache model, FPO-cache made wrong predictions about whether to apply interchange in some cases. For example, when the trip count equals 128, FPO-cache predicts that interchange reduces the number of cache misses by 8224. But simulation result showed that interchange increased the number of cache misses by 3937. Using our cache model, FPO-cache correctly predicted the trend of cache miss increase to 3810. Other benchmarks and other optimizations showed a similar trend.

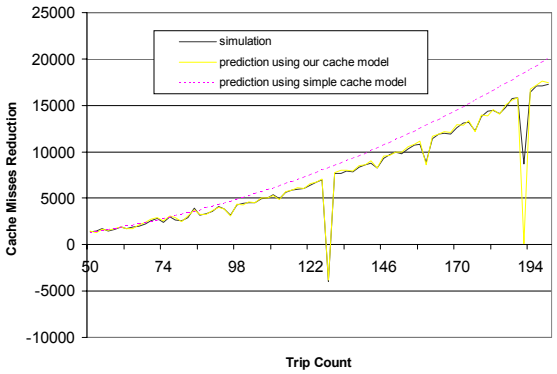

Figure 7. Interchange on *irkernel* with different cache models

Then we computed a prediction accuracy for FPO-cache. If an optimization improves cache performance with the simulation results, and our model predicted that the optimization should be applied, then we consider this to be a *correct prediction*. If the simulation result does not match our predicted result, then it is a *misprediction*. Prediction accuracy captures how often our FPO-cache gives the correct prediction. Table 1 shows prediction accuracy for the single nest benchmark loops with varying trip counts. For each benchmark, the trip count was varied from 50 to 200. From the table, the prediction accuracy ranged from 81.6% to 100% across all benchmarks and optimizations with an average of 97%. Although there is high accuracy across all optimization models, loop reversal has the lowest accuracy. The

reason is that loop reversal has a minimal impact on data cache locality (i.e., the cache miss reduction of applying reversal is very small), and as such, it is difficult to predict its benefit. Although our model chose not to apply loop reversal at those cases, this choice did not degrade the effectiveness of our model because the benefit of applying reversal was so small that it can be ignored (see Figure 6A).

Benchmark	Interchang	Tiling	Reversal
alv	100%	100%	97.4%
irkernel	98.7%	100%	93.4%
lgsi	100%	100%	82%
smsi	100%	100%	86.8%
srsi	100%	100%	86.8%
fsi	100%	97.4%	100%
tomcat3	98.7%	92.1%	93.4%
biquad_N	89.5%	88.2%	100%
gdevcdj	100%	100%	97.4%
lms	97.4%	100%	94.7%
pegwit	100%	100%	81.6%

Table 1. Prediction accuracy for single-loop nest benchmarks

Benchmark	Interchange			Tiling			Reversal		
	A	M	S	A	M	S	A	M	S
adi	2	0	0	2	0	0	2	0	1
aps	1	1	1	1	1	1	3	1	1
eflux	5	5	5	5	1	1	6	2	3
tomcat	6	5	5	6	3	2	9	7	6
vpenta	3	3	3	3	2	2	8	7	7
bmcm	2	2	2	2	2	2	4	3	3

A: Applicable; M: Model Predictions; S: Simulation.

Table 2. Prediction accuracy for multi-loop nest benchmarks.

We also investigated the prediction accuracy of FPO-cache for the benchmarks with multiple loop nests. Table 2 shows the choices made with our models and how the choices compare with actual performance as reported by the simulation framework. For each optimization in the table, there are three columns. The first indicates on how many loop nests in a benchmark an optimization is applicable. The second column indicates the number of loops for which our framework predicts a benefit to applying an optimization. The final column indicates the number of loops in a benchmark in which an optimization should have been applied (i.e., it had an actual performance improvement). As an example, consider loop reversal for *vpenta*. On this benchmark, there are eight loops where reversal could be applied and our framework predicted to apply it in seven cases. The simulation results indicate that the optimization had a benefit on seven loops. In all cases in the table where there are mispredictions, our model selected the same set of loop nests for optimization as the simulation results, except for the one case where there was a misprediction. Although not shown in the table, our model also always made the correct choice for loop unrolling, fusion, and distribution.

3.4. Choosing the Best Optimization

Not only can our model be used to decide whether an optimization should be applied or not, but it can also be used to select among several applicable optimizations. We can use our models to get the predicted benefit of applying each optimization on a loop and then select the one with the maximum benefit. Choosing the best optimization is particularly interesting in our single nest benchmarks when varying the trip count. Here, the trip count (the loop configuration) has a big impact on which optimization is the most beneficial. Figure 8 shows the distribution of optimizations selected for each single nest benchmark with the trip count varied from 50 to 200. The figure shows the percentage of times that a particular optimization was chosen as the best one to apply. When all optimization models predicted a performance degradation (or no benefit), our model decided not to apply any optimization (the "not applying" case in the figure).

For several of the benchmarks, only a couple of choices were made. For example, in *alv*, loop distribution was applied for 11% of the trip counts. For the other 89% of the trip counts, no optimization was applied. The benchmarks *tfsi* and *tomcat3* are interesting since they have three different choices. In *tfsi*, loop reversal, interchange, and tiling were applied, with tiling being applied the most often. For *tomcat3*, loop interchange was most often the best optimization, followed by fusion.

The figure also shows the accuracy of the choices made by our models (in parenthesis below each benchmark name). For most of the benchmarks, the accuracy was above 96%. For the others, such as *smsi* and *srsi*, the accuracy was lower due to mispredictions from our loop reversal model. For example, in *smsi*, the model predicted no benefit to loop reversal, yet there was a very small actual benefit. Notice that from Table 1 we see that reversal had an accuracy of 86%, and as described earlier, the actual benefit was so small that our model did not capture it. Here, the performance improvement due to reversal was minimal.

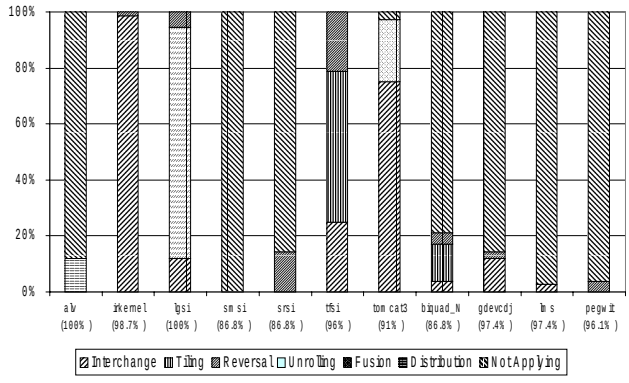

Figure 8 Accuracy and distribution of the most beneficial optimizations for single loop nest benchmarks

3.5. Combining the Optimizations

FPO-cache can be used to help determine the benefits of combining optimizations that are applicable on a code segment. Using our optimization and code models, we can determine the effect of applying one optimization on the code, which would

produce a transformed version of the loop, represented by a code model. This new version could then be used with a model of either another optimization or the same optimization applied initially. The result would be a transformed representation of the code that has the effects of applying both optimizations. This process would continue until a final optimization model is used. The final code model would then be used with a cache model to determine the impact of combining those optimizations.

```
for (I = 1; I <= N; I++)
 for (J=1; J<= N; J++)
  for (K = 1; K<= N; K++)
  CM[K][I]+=AM[K][J]*AM[J][K]+BM[I][J]*BM[J][I];
```

For example, consider the above code (with N=10) that shows three nested loops on which loop interchange can be applied in a number of different ways. The first two loops, I and J, can be interchanged with a benefit of only 0.2%. Although the 2nd and 3rd loops, J and K, can be interchanged, there is a performance penalty of 2.8%. However, by combining both interchanges, we get a new loop nest, J K I, which improves the performance by 12.3%. With our framework we can determine how to combine optimizations and get the best benefits.

We ran experiments on our benchmarks to determine the impact of finding an optimal combination of interchanges on loop nests. With our framework, we found a better interchange combination for *eflux* and *bmcm* than using individual loop interchanges. For *eflux*, applying an optimal combination of loop interchange had a 25.3% performance improvement, while the best single loop interchange had a 18.6% improvement. In the case of *bmcm*, the best combination of loop interchange had a 55% improvement and the best single interchange had a 54% improvement. Thus our framework can be used to determine a combination of the same optimization even if there is no benefit for individual optimizations.

3.6. Compile-time Overhead for Predicting

In embedded systems, achieving the very best performance is paramount and it is worth spending compile time to get better performance. Table 3 shows for several loop benchmarks the compile time overhead (in millisecond) of our tool to predict the impact of optimizations and decide whether to apply an optimization or not. From Table 3, we see that the overhead is dependent on the loop configuration and array references. For example, *irkernel* is a triple loop nest with five references and *srsi* is a double loop nest with 25 references. Their compile-time overhead is higher due to their complexity.

Benchmark	Interchange	Tiling	Reversal
alv (100)	24	29	23
irkernel (100)	2150	2637	2140
lgsi (98)	40	49	38
smsi (124)	118	137	117
srsi (194)	541	630	541
tfsi (42)	8	10	7
tomcat3 (100)	136	160	137
biquad_N (90)	30	36	29
gdevcdj (100)	11	15	11
lms (16)	1	1	1
pegwit (100)	7	10	6

Table 3. Compile-time Overhead for predicting (millisecond)

4. RELATED WORK

Embedded systems have an ever-increasing need for optimizing compilers to produce high quality codes. But there are certain performance problems that remain in current compiler techniques, and in particular, fixed strategies to apply optimizations. Although these problems have been explored in many ways, there is no general, uniform way to effectively address the problems. Previous work has addressed the phase ordering problem in a number of ways. Whitfield and Soffa addressed the problem of applying optimizations by analytically exploring the enabling and disabling properties of optimizations [WS97]. Cooper et al. proposed a biased-random search to find a good order of optimizations [CST01]. Triantafyllis et al. propose an iterative compilation technique, called Optimization-Space Exploration (OSE), to tackle the problem of searching in a large optimization space for other iterative compilation approaches [TVVA03]. Others have combined optimizations to avoid the phase ordering in some cases [CC95]. In optimizing cache behavior, researches have focused on techniques to improve data locality. For instance, Wolf and Lam [WL91] proposed an algorithm that improves the locality of a loop nest based on a mathematical formulation of reuse and locality and a loop transformation theory that unifies various transformations as unimodular matrix transformations. Sarkar [S97] described a transformer that performs automatic selection of loop optimizations using a cost-based framework. Both approaches depend on dependency analysis over the iteration space. McKinley et al. [MCT96] also proposed a compound algorithm to find desirable loop organizations according to a simple cache cost model, which handles the conflict misses in a simpler manner. All of these approaches have cache cost models and algorithms for improving data locality, but it is not clear how these models can be generalized to other optimizations in terms of predicting their impact on performance. Some researchers presented frameworks to combine loop optimizations and array restructuring [CCCM01, KCRB99]. There has also been some research on cache conflict misses. Ghosh et al. [GMM99] described methods for generating cache miss equations that give a detailed representation of cache behavior. G. Rivera et al. [RT98] described some optimizations for eliminating conflict misses. Another technique is to modify the cache configuration for each loop according to its access pattern exhibited by the nest [HKVI02]. Our work differs from the previous work by developing analytic models of optimizations. Our initial focus is optimization impacts on cache performance due to the importance of loop codes and cache for embedded systems. We are currently extending our models to handle other optimizations such as partial-redundancy elimination and code motion.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we described a novel framework, called FPO, for predicting the impact of optimizations on machine resources and performance for embedded processors. With an instance of FPO, namely FPO-cache, we demonstrated the benefits of our framework in tackling several performance problems of optimizations that have been known to the compiler community for years. Using a model of an embedded processor, we showed that prediction can be used to selectively apply a loop transformation based on cache and loop configuration. We also

described and evaluated how FPO-cache can be used to select the best optimization among several applicable ones for a particular code context and combine optimizations. Currently, we are extending FPO to develop models for other resources (e.g., functional units and registers) and other optimizations (e.g., partial redundancy elimination).

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF grant ACI-0203945, and an Andrew Mellon Graduate Fellowship.

7. REFERENCES

[AMP00] Ampro's EnCore Family of Processor-Independent Modules for Embedded Systems, 2000, *http://www.ampro.com/assets/applets/EnCore_Backgrounder.PDF*

[BA97] D.C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *UW Computer Sciences Technical Report 1342, June, 1997*.

[BCK88] M. Berry, D. Chen, P. Koss, D. Kuck and et al. PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications,1988.*

[BGS94] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys, 26(4): 345-420, December 1994*.

[CC95] Click, C. and Cooper, K. D. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS) March 1995.*

[CCCM01] B. Chandramouli, J. Carter, W. Hsieh, and S. McKee. A Cost Framework for Evaluating Integrated Restructuring Optimizations. *International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, September 2001*.

[CST01] K. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *Proceedings of the 2001 LACSI Symposium, Santa Fe, NM, USA, October, 2001.*

[GMM99] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Behavior. *ACM Transactions on Programming Languages and Systems, 21(4): 703-746, July 1999*.

[HKVI02]J. S. Hu, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, H. Saputra, and W. Zhang. Compiler-Directed Cache Polymorphism. *In Proc. of LCTES/SCOPES, June 2002.*

[KCRB99] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers, Vol. 48, No. 2, February 1999.*

[LPM97] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems", *30th Int'l. Symposium on Microarchitecture (MICRO-30)*, December 1997.

[MCT96] K. Mckinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems, 18(4): 424-453, July 1996*.

[MT96] K. McKinley and O. Temam. A Quantitative Analysis of Loop Nest Locality. *Proc. of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems, October 1996*.

[RT98] G. Rivera and C. Tseng. Data Transformations for Eliminating Conflict Misses. *In Proc. of SIGPLAN'98 Conference on Programming Language Design and Implementation, 1998*.

[S97] V. Sarkar, Automatic Selection of high-order transformations in the IBM XL FORTRAN compilers, *IBM Journal of Research and Development, May 1997.*

[SR92] V. Sarkar and R. Thekkath, A General Framework for Iteration-Reordering Loop Transformations. *SIGPLAN Conf. on Programming Lang. Design and Implementation*, 1992.

[T99] Jim Turley, Embedded Processors by the Numbers, *http://www.embedded.com/1999/9905/9905turley.htm*

[TFJ94] O. Temam, C. Fricker and W. Jalby. Cache Interference Phenomena. *In Proc. of SIGMETRICS Conference on Measurement and Modeling Computer Systems, 1994*.

[TVVA03] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler Optimization-space Exploration. *1st International Symposium on Code Generation and Optimization*, March 2003.

[VKIK00] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Kim and W. Ye. A unified energy estimation framework with integrated hardware-software optimizations. *In Proc. of the 27th International Symposium on Computer Architecture, 2000*.

[WL91] M. Wolf and M. Lam, A Data Locality Optimizing Algorithm, *In Proc. of SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada, 1991*.

[WMSW98] D. Weikle, S. Mckee, K. Skadron, and W. Wulf. Caches As Filters: A New Approach To Cache Analysis. *6th Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'98), July 1998, Montreal Canada*.

[WS97] D. Whitfield and M. L. Soffa. An Approach for Exploring Code Improving Transformations. *ACM Transactions on Programming Languages*, *19(6):1053-1084, 1997.*

[ZCW02] W. Zhao, B. Cai, D. Whalley et al., VISTA: A System for Interactive Code Improvement, *ACM Conf. On Languages, Compilers, and Tools for Embedded Systems*, 2002.

[ZMS94] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, DSPstone: A DSP-Oriented Benchmarking methodology, *Proc. of International Conference on Signal Processing Applications and Technology, Dallas, Texas, Oct 1994.*