Cross-loop Reuse Analysis and its Application to Cache Optimizations

Keith Cooper, Ken Kennedy, and Nathaniel McIntosh *

Department of Computer Science
Rice University
Houston, Texas USA

Abstract. In this paper we describe the design of a data-flow framework for detecting *cross-loop reuse*. Cross-loop reuse takes place when a set of data items or cache lines is accessed in a given loop nest and then accessed again within some subsequent portion of the program, usually another outer loop nest. In contrast to *intra-loop* reuse, which occurs during the execution of a single loop nest, cross-loop reuse is hard to analyze using traditional dependence-based techniques. The framework we have constructed is based on a combination of array section analysis (to capture array access patterns at a high level) and data-flow analysis (to deal with intra-procedural control flow). The framework is designed to account for cache size when gathering reuse information, and when used in an interprocedural setting, the framework also provides a mechanism for summarizing the effects of procedure calls.

Cross-loop reuse information can be used to drive a number of transformations that enhance locality and improve cache utilization, including loop fusion and loop reversal. Although these transformations are not new, their impact on cache behavior has not always been possible to predict, making them difficult to apply. As part of this paper we report the results of a comprehensive experimental study in which we apply our techniques to a set of ten programs from the SPEC95 floating point benchmark suite. We were able to obtain modest performance gains overall for several of the programs, based mostly on improvements in cache utilization.

1 Introduction

One of the major trends in computer architecture over the last decade has been the widening gap between processor speed and memory latency. Main memory latencies for modern-day workstations are approaching 100 cycles and beyond, compared with the 1-5 cycle latencies of a decade ago. This huge speed difference has had a number of consequences in the world of computer architecture, chiefly that system designers have had to rely increasingly on cache memories as a means to avoid latency and increase overall memory bandwidth. This is particularly true for shared-memory multiprocessors, since transfers between a processor and

* This work was supported in part by ARPA (Army Contract DABT63-95-C-0115).

memory must pass through additional levels of interconnection, resulting in even longer latencies.

Compiler researchers have also been attacking this problem; they have developed a number of optimizations that seek to enhance cache utilization, including loop interchange, loop tiling, and unroll-and-jam [8, 20] [2]. For most of these techniques, the compiler first analyzes a loop nest to find out what sort of reuse it contains, and then applies transformations to expose or improve the reuse in some way. Accurate information on loop-level reuse is a critical component for these methods; if the compiler can't detect the reuse, then there is no way for it to determine how or when to apply transformations.

Loop-level reuse analysis is most often based on *dependence analysis* [4, 21]. Dependence analysis can provide very detailed information about the memory access patterns within a loop, but applying it to larger regions within a procedure is difficult, especially if the region in question contains control flow or procedure calls. As a result, dependence analysis is not well suited to detecting cross-loop reuse.

Transformations also exist that exploit cross-loop reuse, primarily loop fusion and loop reversal [21]. Because of the shortcomings of existing reuse analysis methods, however, there has been no effective way until now to detect situations where these transformations can be profitably applied.

In this paper we describe a compiler framework for detecting useful cross-loop reuse. Rather than operating on the granularity of individual statements or array references, as is the case with dependence analysis, our framework uses array section analysis to reason about reuse in terms of entire regions within an array. The information derived from array section analysis is then used as input to a data-flow solver, which deals with intra-procedural control flow.

An outline of the remainder of this paper is as follows. In Section 2 we describe the details of our framework. In Section 3 we outline how the information provided by the framework can be applied, and in Section 4 we describe the results of some preliminary experiments using our framework. In Section 5 we discuss related work, and finally in Section 6 we offer our conclusions on this work.

2 Analysis framework

We now describe the details of our analysis. In Section 2.1 we outline some of the key design goals for the problem we are solving, and discuss how we accomplish these goals. In Section 2.2 we discuss array section analysis, an important component of our framework. Section 2.3 introduces the control flow representation used by our data-flow solver. In Section 2.4 we describe the domain over which our data-flow solver operates (i.e. the sets propagated during the analysis). Sections 2.5, 2.6 and 2.7 describe the actual data-flow equations used, along with

[2] Much of this work has been geared towards optimizing scientific programs, since these programs often make relatively poor use of cache memories, and since they are amenable to compile-time methods such as dependence analysis.

their inputs and the details of the solver. In Section 2.8 we discuss incorporation of cache size constraints. In Section 2.9 we briefly cover the algorithmic complexity of some of the operations performed in the framework. Finally, in Section 2.10 we describe how this framework can be used in an interprocedural setting.

2.1 Important design considerations

At a high level, the goal of our framework is to determine, given a point X within the program being analyzed, the set of array regions accessed on all paths that reach X; in many respects, this problem is similar to the well-known compiler optimization problem of "available expressions". We attack this problem, not surprisingly, with data flow analysis.

We do not use a traditional data-flow solver, however, since our problem is unusual in some important respects. First, the problem requires that we explicitly take into account the loop structure of the program, as opposed to treating all control flow in an identical fashion. This is due to the fact that a given subscripted reference may access a different region in an array depending on how much of the surrounding loop context is taken into account (see section 2.2).

A second aspect of the problem that complicates the analysis is that we want to take into account the size of the cache. The tools we have developed for this sub-task are difficult to combine with traditional flow analysis techniques. The solver we use allows us to develop a framework for detecting reuse without considering cache size constraints, and then factor in the cache size if necessary (see section 2.8).

2.2 Array sections

Array section analysis is a technique for summarizing the region(s) within an array that are accessed during some portion of the program [2, 5, 7, 15]. These summary representations provide a compact way of capturing the array access patterns, making them attractive for applications in which large portions of the program need to be considered. Our particular implementation represents array accesses using *Data Access Descriptors* [3], or "DAD"s.

The region within an array accessed by a given subscripted reference depends on the context that surrounds the reference. For example, consider the reference `a(i,j)` in Figure 1. If we consider this array reference in isolation, then it can be thought of as accessing the single element at `a(i,j)`. If we take the `do i` loop into account, then the reference can be thought of as accessing the vector `a(1:n,j)`, and so on.

We use the term *array section at level M* to refer to the region within an array that is accessed at a given loop level. More formally, given a subscripted reference nested within N loops, the array section at level M for the reference is the region accessed within the array when A) the loop induction variable(s) at levels M-1 and below are viewed as invariants, and B) the induction variables at levels M and above are allowed to vary.

2.3 Control flow representation

Rather than using a standard control flow graph (CFG), this framework uses an *interval-flow graph*, developed by Reinhard von Hanxleden and Ken Kennedy [14]. This allows us to take the loop structure of the program into account explicitly. The interval-flow graph (IFG) can be constructed by starting with a normal CFG and then partitioning the nodes and edges in the graph into categories based on Tarjan interval analysis [18] [3].

A Tarjan interval $T(h)$ is a set of CFG nodes that corresponds to a loop within the program, where h is a unique header node (with $h \notin T(h)$). Intuitively, $T(h)$ together with h form a strongly connected region within the CFG. When $x \in T(h)$, we say that $\textsc{Header}(x) = h$.

Each interval-flow graph $G = (N, E)$ has a unique root node, $\textsc{Root}$, that can be viewed as the header node for the interval corresponding to the entire procedure. We define $\textsc{Level}(n)$ as the loop nesting level of node n, counting from the outside in, where $\textsc{Level}(\textsc{Root})$ is defined as 0. For a given interval $T(n)$, we define $\textsc{Children}(n)$ as the set of nodes $\{\ p \in T(n) : \textsc{Level}(p) = \textsc{Level}(n)+1\ \}$, that is, the nodes in the interval headed by n that are immediate descendants of n. Figure 1 shows an example program fragment along with its interval-flow graph.

Each edge $(x \to y)$ in the IFG is placed into one of the following categories:

> **Entry:** iff $y \in T(x)$; this corresponds to an edge from an interval header x to a node within the interval.
>
> **Cycle:** iff $x \in T(y)$; this corresponds to an edge from within an interval back to the header of that interval.
>
> **Jump:** iff $\exists h : x \in T(h),\ y \notin \{\ T(h) \cup h\ \}$; this corresponds to a jump out of a loop (i.e. an edge from a node in one interval to a node outside the interval that is not the header node).
>
> **Flow:** iff $\forall h : x \in T(h) \iff y \in T(h)$; this corresponds to an intra-interval edge (i.e. an edge that is none of the above).

To refer to the predecessors and successors of a given node, we use the following terminology:

> **$\textsc{Preds}(n)$:** The set of nodes $\{\ x\ :\ \exists e \in E,\ e = (x, n)\ \}$
>
> **$\textsc{Succs}(n)$:** The set of nodes $\{\ x\ :\ \exists e \in E,\ e = (n, x)\ \}$

The $\textsc{Preds}(n)$ and $\textsc{Succs}(n)$ notation may be further qualified by adding a superscript containing the desired edge type. For example, $\textsc{Preds}^F(n)$ is the set of nodes that are at the source of a $\textsc{Flow}$ edge whose sink is n.

[3] The interval-flow graph is not be confused with interval-based data-flow analysis. With the interval-flow graph, there is no notion of collapsing intervals in the CFG into single nodes, as in interval-based data-flow analysis.

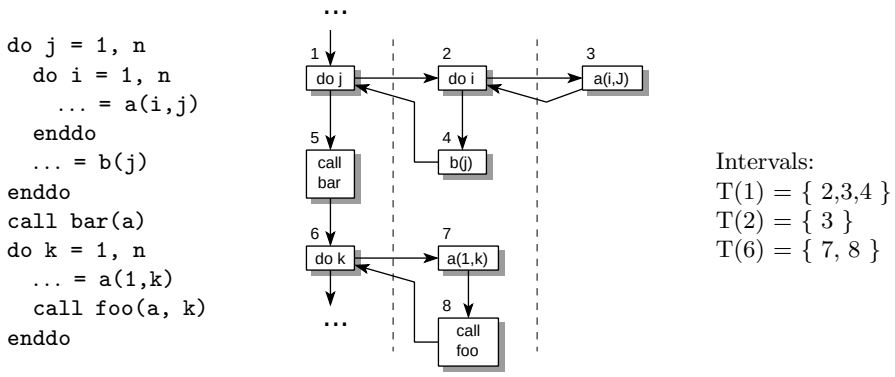

Fig. 1. Example sub-program with interval-flow graph

After construction of the IFG, the graph is post-processed to insure that each interval has at most one CYCLE edge, i.e. for each non-empty interval $T(h)$ there exists a unique $n \in T(h)$ such that $(n, h) \in E$. This sometimes requires the insertion of *synthetic* nodes and edges; see [13] for details.

In addition, we define two types of partial orderings on N, as follows:

FORWARD/BACKWARD**:** Given a FLOW/JUMP edge (m, n), a FORWARD order visits m before n, whereas a BACKWARD order visits m after n.

UPWARD/DOWNWARD**:** Given $m, n \in N$ such that $m \in T(n)$, an UPWARD order visits m before n, whereas a DOWNWARD order visits n before m.

These two orderings are orthogonal and may be combined (for example, a FORWARD and DOWNWARD order).

2.4 Data-flow universe

The universe for this data-flow problem consists of sets of array section descriptors, where each descriptor is composed of the name of the array in question and a symbolic representation of the region accessed within the array. We call these sets *reuse summary sets*. An example of a reuse summary set might be

$$\{\ a(1:10),\ a(75:n),\ b(1:n)\ \}$$

As can be seen from the set above, a reuse summary set may contain more than one region within a given array. The interpretation of the set depends on the context; it may represent a set of array sections available on entry to a given block, generated within a given loop, etc.

2.5 Initial information

For each basic block n that contains one or more array references, we compute the set $GEN_{INIT}(n)$. This reuse summary set contains an array section for each array reference within the block. The sections initially assigned to $GEN_{INIT}(n)$ are computed with respect to the innermost loop level. During the analysis, when the section is propagated up out of an enclosing loop, the region of the section is reconstructed to take the loop in question into account.

During the computation of the initial set for a block, we attempt to coalesce sections that are adjacent or identical. For example, we might try to collapse the set { `a(i,j), a(i+1,j)` } into { `a(i:i+1,j)` }. This coalescing is only performed when it will result in no loss of precision, however.

$$GEN_{IN}(n) = \bigwedge_{p \in PRED^F(n)} \{\ GEN_{OUT}(p)\ \} \tag{1}$$

$$GEN_{LOC}(n) = \begin{cases} GEN_{OUT}(\textsc{Lastchild}(n)) & \text{if } n \text{ is an interval header} \\ GEN_{INIT}(n) & \text{otherwise} \end{cases} \tag{2}$$

$$GEN_{OUT}(n) = GEN_{IN}(n) \vee GEN_{LOC}(n) \tag{3}$$

$$REACH_{IN}(n) = \begin{cases} REACH_{IN}(\textsc{Header}(n)) & \text{if } \textsc{Preds}^E(n) \neq \emptyset \\ \bigwedge_{p \in PRED^F(n)} \{\ REACH_{OUT}(p)\ \} & \text{otherwise} \end{cases} \tag{4}$$

$$REACH_{OUT}(n) = REACH_{IN}(n) \vee GEN_{LOC}(n) \tag{5}$$

Fig. 2. Reuse equations

2.6 Reuse equations

Our goal for the flow analysis is to compute the set $REACH_{IN}(n)$ for all of the nodes in the IFG. The set $REACH_{IN}(n)$ corresponds informally to the set of array sections that reach node n on all paths from Root to n. We compute $REACH_{IN}$ using the equations in Figure 2; evaluation of the equations is controlled by the algorithm in Figure 3. All sets are initially empty (with the exception of GEN_{INIT}).

Intuitively, $GEN_{IN}(n)$ corresponds to the set of sections accessed by the nodes prior to n within the interval that contains n. $GEN_{LOC}(n)$ corresponds to the set of sections accessed within n (if n is not a loop header) or the sections accessed within the interval headed by n (if n is a loop header). $GEN_{OUT}(n)$ combines GEN_{IN} and GEN_{LOC} to form the set of sections that flow out of

n (taking into account only the nodes in the interval containing n). The GEN values are computed starting with innermost loops and working outwards.

$REACH_{IN}(n)$ corresponds to the set of sections reaching n from within n's interval and (possibly) from some previous loop outside n's interval. $REACH_{OUT}(n)$ is the set of sections flowing out of n, where the sections may be generated locally or they may reach n from some previous loop somewhere in the program.

---

procedure *ComputeReuse*

inputs: interval-flow graph $G = (N, E)$
$GEN_{INIT}(n)$ for all $n \in N$
outputs: $REACH_{IN}(n)$ and $REACH_{OUT}(n)$ for all $n \in N$

begin
  forall $n \in N$ in UPWARD and FORWARD order:
    compute equations 1, 2, and 3
  forall $n \in N$ in DOWNWARD and FORWARD order:
    compute equations 4 and 5
end

Fig. 3. Procedure for computing reuse equations

2.7 $\bigvee$ and $\bigwedge$ operators

In the equations above, the $\vee$ and $\wedge$ operators play an important role. The $\wedge$ operator is used to merge together sets of sections at join points, and the $\vee$ operator is used to merge together local information with incoming information (i.e. models the effects of passing through a block).

The left hand side of Figure 4 illustrates a situation where the $\wedge$ operator would be applied. In this case, the analysis needs to merge together the sections reaching the node `X` from its two predecessors `M` and `N`, taking into account the fact that the particular path to `X` is unknown.

The right hand side of Figure 4 shows a situation where the $\vee$ operator is needed. The sections flowing out of `X` need to be combined with the sections locally generated at `Y`, but in this case we know that flow of control *must* reach `Y` if it reaches `X`.

2.8 Incorporating cache constraints

If the goal is to predict reuse without regard to cache size or other resource constraints, then we can implement the $\vee$ operator as set union and the $\wedge$ operator as set intersection. The resulting information, however, does not give any hint as to distance between successive uses of array data (only that some set of array locations are reused). In many situations we would like to know whether

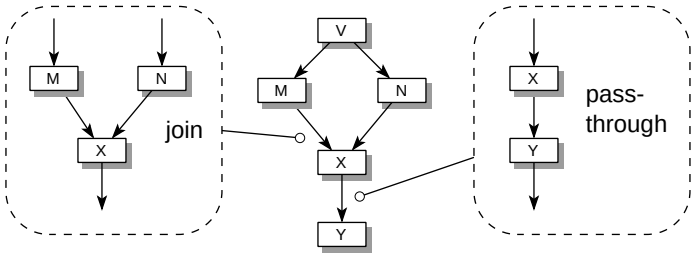

Fig. 4. Control flow

the distance between the successive uses is small enough that the reused items will be found in cache, for some fixed cache size. This section describes how we modify our framework to take cache size and organization into account.

Array section age: First, we introduce the concept of the "age" of an array section with respect to a particular point in the program. We define the age of a given section as the number of cache misses that have taken place since the first element of the section was brought into the cache.

During the analysis, we associate age values with each of the sections in a reuse summary set. When an array section is first added to a reuse summary set (corresponding to the point where it is first brought into the cache) we assign it an initial age value based on its volume. As the section is propagated to other points in the program, other accesses will start to displace the section from the cache; when this happens, the age of the section is incremented. Eventually the age of the section reaches a cutoff, at which point we consider the section "dead" (i.e. totally displaced from the cache), and it is eliminated from its reuse summary set.

In order for this scheme to work, we need to have a mechanism for determining the number of cache lines accessed by an array reference within a loop; this is in fact a research problem all by itself [10]. Our approach is to estimate the volume of the DAD for the reference, using a simple technique similar to the RefCost algorithm developed by Carr, McKinley, and Tseng [8].

Cache organization: Our framework is not equipped to predict cache conflicts due to limited associativity; we instead conservatively assume that cache conflicts will reduce the amount of reuse that takes place by a fixed factor. We currently estimate the "effective" size of the cache (used in the analysis) by multiplying the actual cache size by $1 - \frac{1}{2^S}$, where S is the set associativity of the cache.

$\vee$ operator for finite caches: For the finite cache case, we use a more sophisticated $\vee$ operator that models the cache effects when execution passes through a given node (shown in Figure 5). Given the set of sections flowing into block N (IN) and the set of sections accessed locally within N and N's descendants

(LOC), this new $\vee$ operator computes the OUT set, taking into account the level of reuse and the size of the cache. The algorithm is based on the observation that a given section $S \in LOC$ will cause cache misses only if it is not contained in some section $S' \in IN$.

---

procedure *FiniteCache-*$\vee$

inputs: IN (incoming reuse summary set)
LOC (reuse summary set for locally accessed data)
outputs: OUT (outgoing reuse summary set)

begin
  `volume` $\leftarrow 0$
  $R \leftarrow \emptyset$
  forall `x` $\in LOC$:
    if $\exists$ `y` $\in IN$ such that `x` is contained in `y` then
      $R \leftarrow R \cup \{$ `y` $\}$
    else
      `volume` $\leftarrow$ `volume` + (cache line volume of `x`)
    endif
  endfor
  $OUT \leftarrow LOC \cup (IN$ - $R)$
  forall `x` $\in OUT$:
    increment age value of `x` by `volume`
    remove `x` from OUT if age exceeds cache size cutoff
  endfor
end

Fig. 5. $\vee$ for finite cache case

$\wedge$ operator for finite caches: We also modify the $\wedge$ operator when estimating reuse for a finite cache. Set intersection is still the basis for the operator, but the age values of the sections must be adjusted as well. In particular, when the reuse summary sets for two incoming paths are merged together by the $\wedge$ operator, we may encounter an array section that appears in both sets, but has a different age value in each one. In this case, the $\wedge$ operator chooses the larger of the two ages for the section in question when forming the result.

For example, consider the graph fragment in Figure 4. Suppose that we are applying the $\wedge$ operator to the sections reaching block X from its predecessors M and N, and suppose that blocks V, M, and X contain array accesses (each to a different array), and that N contains no accesses. When we apply the $\wedge$ operator, both input sets will contain the section from V (we assume here that the number of accesses in M is relatively small), however the age of the sections from V that arrive at X along the edge $M \rightarrow X$ will be larger than the age of the

corresponding sections flowing through the edge $N \rightarrow X$ (due to the additional data brought into the cache in M). The $\wedge$ operator selects the larger age, in order to be conservative.

2.9 Complexity

Generating the GEN_{INIT} set for each block requires that we build a DAD for each array reference in the procedure. This process requires $O(D^2 * N)$ time per reference, where D is the number of dimensions of the array and N is the nesting depth of the reference. Each DAD takes $O(D^2)$ space, and most operations involving DADs (union, intersection, containment, comparison) take $O(D^2)$ time.

The flow analysis framework itself considers each node and edge in the IFG exactly twice. The $\wedge$ and $\vee$ operators for the unlimited-cache case are linear in the number of sections in the sets being operated on, but for the finite-cache case, $\vee$ takes $O(N^2)$ time in the worst case (where N is the number of sections in each set), since since each section in the set may have to be compared with every other section.

In practice, we have found that the time required by the framework is comparable to the time that it takes to perform dependence analysis for the procedure.

2.10 Interprocedural analysis

Array section analysis was originally conceived of as a means of summarizing the effects of procedure calls within loops; as a result, it is relatively straightforward to extend our framework to work in an interprocedural setting.

When invoked in the context of whole-program analysis, we use the following strategy. We analyze procedures starting with the leaves in the call graph and working up to the root, visiting a procedure only after all of the procedures it calls have been visited. When a procedure call is encountered within the IFG of the subroutine being analyzed, we take the previously computed $REACH_{OUT}$ set for the callee and use it as the GEN_{LOC} set for call (applying array reshapes if necessary, and translating the summary into the name space of the caller by substituting actuals for formals, etc).

Summarizing call sites in this fashion is generally feasible only if the framework is being run with cache size constraints taken into account (without the size constraints, the reuse summary sets grow very large in the upper regions of the call graph).

Currently we are restricted to propagating information upwards in the call graph; we do not, for example, take advantage of context within calling routines to reason about reuse within a callee. Our analysis does not currently handle programs whose call graphs contain cycles.

3 Applications of cross-loop reuse information

This section describes some of the ways in which cross-loop reuse information can be used by a compiler. It should be noted that exploiting cross-loop reuse

information tends to be more difficult than exploiting loop-level reuse information; the larger the region over which the reuse is taking place, the more more obstacles that must be overcome if restructuring is to be applied.

3.1 Locality-enhancing loop transformations

Cross-loop reuse can be used to predict the profitability of locality-enhancing transformations involving pairs of adjacent loop nests. These transformations include (but are not limited to) loop fusion and loop reversal.

Loop fusion: Loop fusion is the dual of loop distribution; it combines two adjacent loops with identical headers into a single loop. Loop fusion has the potential to convert cross-loop reuse into loop-level reuse, which can greatly enhance cache utilization.

Our framework can supply enough information to cheaply predict the profitability of loop fusion. The compiler can examine the $REACH_{IN}$ set for a given loop nest to see what sections reach the loop. If the intersection of the $REACH_{IN}$ set with the GEN_{LOC} set of the loop is sufficiently large and the sections in the $REACH_{IN}$ set originate from the immediately preceding loop nest, then loop fusion will be profitable (the degree of profitability will be dependent on the volume of the intersection).[4] Once it is established that the transformation is profitable, then the compiler can apply the more costly dependence-based techniques to determine whether fusion is safe [21].

Loop reversal: A weaker but slightly more widely applicable technique is loop reversal. This optimization can provide benefits only in proportion to the size of the cache, thus it works best for very large (presumably secondary or tertiary) caches. Consider two consecutive outer loops that both access a single large vector (i.e. larger than will fit in the cache). Even though the vector is reused, there is no cache reuse, since when the second loop begins execution, the first elements of the vector have long since been flushed from cache. However if we reverse the second loop, then the last elements of the vector from the previous loop are likely to still be in cache. This optimization relies on the *traversal order* component of the DAD representation, which captures the direction and stride of the access in each array dimension (see [3] for the details). One advantage of loop reversal is that the loops to be optimized do not have to be directly adjacent (there may be intervening code, provided that it does not destroy the reuse between the loops).

3.2 Transformation selection

Even if the compiler can cheaply predict when a transformation is going to be profitable, there still remains the problem of deciding the sequence of transformations to apply within a procedure. A given loop nest may be optimized in

[4] This requires that we tag each section with the ID of the loop in which it originated.

several different ways (fused with its predecessor or with its successor, for example). Selecting the optimal set of transformations is a very difficult problem; optimizing temporal locality using loop fusion alone is NP-hard [16]. As a result, the compiler must resort to heuristics to choose the set of transformations to apply.

4 Experimental results

In this section, we report the results from an experimental study, in which we apply our techniques to ten programs from the SPEC95 floating point benchmark suite [19]. Our experimental infrastructure consists of a Fortran transformation engine, including the cross-loop reuse analysis framework, and an execution-driven simulator for gathering instruction counts and cache statistics.

4.1 Compiler

The phases in our compiler are shown in Figure 6. The compiler operates in a source-to-source fashion, reading and writing Fortran code. All of the analysis and transformation steps shown are performed automatically.

Phase	Remarks
1. Front end processing	read and typecheck Fortran source build AST (abstract syntax tree)
2. Local analysis	build IFG for each procedure compute GEN_{INIT} for all $n \in$ IFG
3. Cross-loop analysis	run cross-loop framework for each function
4. Transformations	apply loop reversal and loop fusion based on cross-loop reuse info
5. Output	generate transformed Fortran source

Fig. 6. Compilation stages

Our transformation selection procedure is as follows. For each loop nest N, we create a hash table (the "reuse score table") whose entries are tuples of the form $\langle L, R \rangle$, where L is a loop ID number and R is an estimate of the number of cache lines reused from loop L. We generate the reuse score table for a loop N while computing the value of equation 3 in Figure 2; when a section $x \in GEN_{IN}(N)$ intersects with a section $y \in GEN_{LOC}(N)$, we compute the volume of the intersection of x and y, determine the loop L' in which x originated, and update the L' entry in N's table. We calculate the total score for a loop by summing the values of all of the entries in its reuse score table.

We then use a greedy algorithm to select the loops to optimize; we first consider the loop nest with the highest reuse score value, optimize it if possible, then consider the loop with the next highest score, and so on. We first apply loop

fusion, then loop reversal. In order to concentrate our results primarily on cache effects, we limited loop fusion to the outermost loop in each pair of adjacent loop nests.

4.2 Simulator

Our cache simulator is based on the SPARC Performance Analysis Toolkit; it is layered on top of the tool `shade` [9]. `Shade` provides an extensible mechanism for writing execution-driven simulators; it operates by interpreting a SPARC executable and passing a trace of the instructions to a user-written trace analyzer. In our case, the trace analyzer counts instructions and simulates a particular cache configuration.

After the source-to-source transformer is run, the target programs are instrumented with calls to runtime routines to tag outer loop nests and to demarcate the regions of the program's address space containing array data. The instrumented programs are then compiled using the Sun `f77` Fortran compiler (version SC4.0 18/Oct/95).

The simulator deals with *data* cache behavior only; it does not simulate an instruction cache. Simulated cache characteristics were as follows. We used a 64Kbyte, 4-way set-associative L1 cache with a line size of 32 bytes, and a 1024Kbyte L2 cache, also 4-way set-associative with a line size of 32 bytes. An LRU replacement policy was used within each cache set. Both caches are write-back, with an allocate-on-write-miss policy.

Program	Functions	Lines	Data (KB)	Runtime (secs)
applu	16	1869	32,309	25
apsi	96	4576	9,381	61
fpppp	38	2408	454	6
hydro2d	42	1597	8,630	182
mgrid	14	448	7,464	192
su2cor	36	1700	23,806	551
swim	6	254	14,392	15
tomcatv	1	119	14,407	181
turb3d	23	1294	25,360	337
wave5	104	7036	41,403	65

Fig. 7. Program characteristics

4.3 Benchmark programs

Figure 7 gives some of the summary characteristics of the programs we used for our experiments. "Functions" is the number of procedures in the program; "Lines" is the number of non-comment source lines. "Data" is the total size of all

the arrays used by the program, in kilobytes. "Runtime" shows the approximate wall clock running time on an unloaded SPARCStation 10 with 256 megabytes of memory. The "training" input files were used for these runs, in order to yield more reasonable simulation times [19].

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	329,240	66,962	5,962	4,731	1,231	487,340
apsi	2,333,872	362,521	5,427	5,413	4	2,388,462
fpppp	240,043	23,771	38	36	2	240,583
hydro2d	5,543,080	1,186,888	218,461	16,091	202,371	23,917,370
mgrid	12,566,952	3,445,822	119,285	102,698	16,587	15,086,762
su2cor	23,341,356	6,425,969	1,215,323	1,024,815	190,508	50,735,226
swim	481,252	119,409	8,784	21	8,763	1,270,132
tomcatv	6,398,033	1,647,655	172,041	26,139	145,901	19,790,523
turb3d	14,453,171	2,568,151	84,751	34,275	50,475	19,338,681
wave5	2,700,459	564,179	38,124	30,988	7,106	3,650,179

Fig. 8. Simulation data for original programs [thousands]

4.4 Results

Figure 8 gives the raw instruction counts and cache metrics for the original untransformed programs. All numbers are in thousands. The "Instructions" column contains the total dynamic instruction count for the program. The L1 and L2 cache metrics are for accesses to array data only (i.e. they exclude accesses to scalars, compiler-generated spill code, etc). The "P-cycles" term in the final column is an approximation of the overall execution time of the program; it combines the total instruction count with the projected stalls due to cache misses. It is computed as follows:

$$\text{P-Cycles} = \text{IC} + (M_1 * P_1) + (M_2 * P_2)$$

where "IC" is the total instruction count, M_k is the total number of misses at level k, and P_k is the additional miss penalty at level k. For the purposes of our study, we assume a level 1 miss penalty of 10 cycles, and an additional level 2 miss penalty of 80 cycles (a miss to main memory takes a total of 90 cycles, in other words).

Figures 9, 10, and 11 show the results for the transformed programs using purely static reuse analysis. Figure 9 gives a summary of the specific transformations applied to each of the programs. The numbers show for "candidates" indicate the total number of loops in the program that were determined to be legal candidates for fusion or for reversal. Figure 10 shows the raw data for the transformed programs (again, all numbers are in thousands). Figure 9 compares the transformed programs to the original programs in each category, showing

		fusion		reversal	
Program	loops	fused	candidates	reversed	candidates
applu	168	2	4	20	111
apsi	298	1	2	5	150
fpppp	39	0	0	0	6
hydro2d	163	3	14	5	136
mgrid	57	0	1	7	36
su2cor	118	0	0	3	47
swim	24	0	4	0	22
tomcatv	16	0	2	0	8
turb3d	64	0	0	0	33
wave5	377	14	27	27	212

Fig. 9. Transformation summary

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	330,029	67,138	5,814	4,582	1,232	486,729
apsi	2,333,872	360,505	5,411	5,399	4	2,388,302
fpppp	240,042	23,518	39	37	2	240,592
hydro2d	5,542,709	1,197,341	218,262	18,518	199,744	23,704,849
mgrid	12,561,752	3,448,549	116,891	100,306	16,577	15,056,822
su2cor	23,341,354	6,423,729	1,217,500	1,027,017	190,482	50,754,914
swim	481,490	119,409	8,784	21	8,763	1,270,370
tomcatv	6,397,215	1,647,651	172,044	26,143	145,901	19,789,735
turb3d	14,668,168	2,555,806	95,870	45,379	50,490	19,666,068
wave5	2,686,798	559,578	38,891	31,779	7,081	3,642,188

Fig. 10. Simulation data for transformed programs [thousands]

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	0.23	0.26	-2.48	-3.14	0.08	-0.12
apsi	0	-0.55	-0.29	-0.25	0	0
fpppp	0	-1.06	2.63	2.77	0	0
hydro2d	0	0.88	-0.09	15.08	-1.29	-0.88
mgrid	-0.04	0.07	-2.00	-2.32	-0.06	-0.19
su2cor	0	-0.03	0.17	0.21	-0.01	0.03
swim	0.04	0	0	0	0	0.01
tomcatv	-0.01	0	0	0.01	0	0
turb3d	1.48	-0.48	13.11	32.39	0.02	1.69
wave5	-0.50	-0.81	2.01	2.55	-0.35	-0.21

Fig. 11. Percent change between original and transformed

percent change; as can be easily seen, there is very little overall change in program performance.

After a closer inspection of the results, we found that the framework appeared to be missing a number of important transformation opportunities, due primarily to situations loop bound values could not be determined at compile time (forcing the compiler to conservatively assume no reuse). To explore this hypothesis, we ran a new set of experiments in which the compiler incorporated profiling data into the analysis framework.

Profiling data was gathered by running each program on the given input file and gathering the minimum, maximum, and average value for the loop bounds, loop step, and overall trip count. For loops whose bounds were constant at run-time, the compiler substituted in the bounds from the profiling data when computing the GEN_{LOC} set for the loop in question. It should be emphasized that profiling data was only used in the portions of the analysis that determine profitability of transformations, not the safety of the transformations.

Figures 12, 13, and 14 show the results with the profiling information. By using profiling data, the compiler was able to compute more accurate estimates of the number of cache lines accessed in each loop nest, and was able to detect reuse in situations where previously it had to conservatively assume no overlaps. The number of loops fused went up from 20 to 24, and the number of reversed loops went up from 67 to 84. The results in Figure 14 indicate that two programs, `tomcatv` and `hydro2d`, showed improvements in performance as a result of the transformations. In both cases, the speedup is due to improved cache behavior; both programs show significant reductions in both L1 and L2 misses overall.

5 Related work

A number of researchers have developed compiler techniques useful for improving cache behavior [1, 6, 8, 17, 20]. Almost all of these techniques apply to individual loop nests, however, and are not designed to detect or exploit cross-loop reuse. Two exceptions are *loop fusion* and *affinity regions*.

Kennedy and McKinley have proposed using loop fusion to improve locality and cache behavior [16]. In a subsequent study, McKinley, Tseng, and Carr included loop fusion in their repertoire of transformations for an experimental study on compiler cache optimizations [8]. This study used dependence analysis to test for the profitability of loop fusion; loop reversal was not used as a locality-enhancing transformation.

Affinity regions are a mechanism that allows a compiler or user to give locality-improving hints to the loop scheduler for a parallel program running on a shared-memory multiprocessor. By placing a set of consecutive parallel loops within an affinity region, the user or compiler is informing the loop scheduler that cross-loop reuse exists and that it should try to assign iterations to processors in such a way that the reuse is preserved. Compile-time identification of affinity regions was proposed by Appelbe et al [1].

		fusion		reversal	
Program	loops	fused	candidates	reversed	candidates
applu	168	2	4	20	111
apsi	298	1	2	5	150
fpppp	39	0	0	0	6
hydro2d	163	5	14	20	136
mgrid	57	0	1	7	36
su2cor	118	0	0	5	47
swim	24	1	4	1	22
tomcatv	16	1	2	0	8
turb3d	64	0	0	0	33
wave5	377	14	27	26	212

Fig. 12. Transformation summary (with profile)

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	329,562	67,122	5,813	4,584	1,229	486,012
apsi	2,333,873	362,635	5,289	5,277	4	2,387,083
fpppp	241,000	23,518	39	37	2	241,550
hydro2d	5,581,854	1,195,747	209,575	30,474	179,101	22,005,684
mgrid	12,561,753	3,446,434	118,673	102,110	16,563	15,073,523
su2cor	23,342,471	6,423,530	1,217,698	1,027,185	190,514	50,760,571
swim	480,793	119,411	8,783	29	8,754	1,268,943
tomcatv	6,396,957	1,663,997	155,700	23,096	132,603	18,562,197
turb3d	14,453,171	2,567,901	84,701	34,225	50,476	19,338,261
wave5	2,705,524	559,560	38,909	31,797	7,081	3,661,094

Fig. 13. Simulation data for transformed programs, with profile [thousands]

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	0.09	0.23	-2.49	-3.10	-0.16	-0.27
apsi	0	0.03	-2.54	-2.51	0	-0.05
fpppp	0.39	-1.06	2.63	2.77	0	0.40
hydro2d	0.69	0.74	-4.06	89.38	-11.49	-7.99
mgrid	-0.04	0.01	-0.51	-0.57	-0.14	-0.08
su2cor	0	-0.03	0.19	0.23	0	0.04
swim	-0.09	0	-0.01	38.09	-0.10	-0.09
tomcatv	-0.01	0.99	-9.49	-11.64	-9.11	-6.20
turb3d	0	0	-0.05	-0.14	0	0
wave5	0.18	-0.81	2.05	2.61	-0.35	0.29

Fig. 14. Percent change between original and transformed (with profile)

Our data-flow framework resembles that of Gross and Steenkiste [11]. However their framework is geared towards finding parallelism as opposed to detecting useful reuse for cache optimizations. Our techniques are also similar to those developed by Gupta, Schonberg, and Srinivasan for optimizing communication placement for programs running on distributed-memory multiprocessors [12].

6 Conclusions

In this paper we have presented a framework for predicting cross-loop reuse. The framework combines two existing tools: array section analysis and data-flow analysis. By using array sections, we can exploit the characteristics of the program's array access patterns without resorting to potentially costly procedure-wide dependence analysis. By using data-flow analysis, we can gracefully handle intra-procedural control flow. Our framework also provides a mechanism for taking into account a fixed cache size when predicting reuse, should this be required.

This work opens up the possibility of systematically applying cross-loop transformations to improve cache utilization, since it provides a means of cheaply predicting the profitability of loop fusion and particularly loop reversal.

Our experimental results suggest that for programs that exhibit cross-loop reuse, our analysis framework is able to detect this reuse at apply cross-loop transformations to exploit it. For programs running on uniprocessors, our results translate into modest improvements in overall execution time; we would expect to see more significant gains for shared-memory multiprocessors, where small increases in second-level cache utilization can sometimes result in significant performance improvements.

7 Acknowledgements

The authors would like to thank Kathryn McKinley, Jerry Roth, Taylor Simpson, and Phil Schielke for their comments on earlier drafts of this paper.

References

1. B. Appelbe and B. Lakshmanan. Program transformations for locality using affinity regions. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.
2. V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD thesis, Dept. of Computer Science, Rice University, May 1989.
3. V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.
4. U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.
5. M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

6. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.
7. D. Callahan, J. Cocke, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1988.
8. S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.
9. R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12; UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and University of Washington, 1993.
10. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.
11. T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.
12. M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.
13. R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems*. PhD thesis, Dept. of Computer Science, Rice University, December 1994.
14. R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.
15. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
16. K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.
17. W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, MA, October 1992.
18. R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.
19. J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.
20. M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
21. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

This article was processed using the LaTeX macro package with LLNCS style