Combining Loop Transformations Considering Caches and Scheduling

Michael E. Wolf, Dror E. Maydan and Ding-Kai Chen
Silicon Graphics, Mounain View, CA

Abstract

The performance of modern microprocessors is greatly affected by cache behavior, instruction scheduling, register allocation and loop overhead. High level loop transformations such as fission, fusion, tiling, interchanging and outer loop unrolling (e.g., unroll and jam) are well known to be capable of improving all these aspects of performance. Difficulties arise because these machine characteristics and these optimizations are highly interdependent. Interchanging two loops might, for example, improve cache behavior but make it impossible to allocate registers in the inner loop. Similarly, unrolling or interchanging a loop might individually hurt performance but doing both simultaneously might help performance.

Little work has been published on how to combine these transformations into an efficient and effective compiler algorithm. In this paper, we present a model that estimates total machine cycle time taking into account cache misses, software pipelining, register pressure and loop overhead. We then develop an algorithm to intelligently search through the various possible transformations, using our machine model to select the set of transformations leading to the best overall performance. We have implemented this algorithm as part of the MIPSPro commercial compiler system. We give experimental results showing that our approach is both effective and efficient in optimizing numerical programs.

Keywords: *cache tiling, outer loop unrolling, loop interchange, fission, fusion, instruction scheduling.*

1.0 Introduction

High level loop transformations such as fission, fusion, tiling, interchanging and outer loop unrolling are individually well known to improve cache utilization, instruction scheduling and register allocation.[1] Difficulties arise because the interactions between these optimizations and between these machine characteristics are complicated and highly machine dependent.

Consider optimizing the following loop nest for a superscalar processor such as the MIPS R10000 having a 32KB on-chip cache with a 32B line size.

```
subroutine nest(a, b, c)
real*8 a(1000)
real*8 b(1000,1000),c(1000)
DO j = 1, 1000
   DO i = 1, 1000
      a(j) = a(j) + b(j, i) * c(j)
   END DO
END DO
end
```

Let's start by considering only the interchange transformation. The decision of whether to interchange or not cannot be made without intimate knowledge of both the cache system and the processor details. If we do not interchange, we expect a cache miss each iteration because the *b* reference is not stride one. If we interchange, each reference is stride one, so we expect only one miss for *b* every four iterations.

But the cache performance is only part of the story. Without interchange, each iteration requires only one multiply-add and one load. Assuming that we can reassociate reductions and ignoring cache misses this will take one cycle per iteration on the MIPS R10000. If we interchange, we have to do two extra loads and one extra store per iteration. Since the R10000 has only one load/store unit, the interchanged loop will take four cycles per iteration, ignoring cache misses.

Depending on the cache miss penalty, interchange might help or might hurt performance. Adding other transformations into the mix further complicates matters. We can both interchange and outer loop unroll our example as follows.

1. We use the term *outer loop unrolling* instead of *unroll-and-jam* because when loop bounds are not constant, the specific transformation applied is not an *unroll* followed by a *jam*, but a more complicated transformation that has the equivalent effect on scheduling. Likewise, we use the term *tiling* rather than *strip-mine and interchange* for the same reason.

```
DO i = 1, 1000,4
   DO j = 1, 1000
      a(j) = a(j) + b(j,i)*c(j)
      a(j) = a(j) + b(j,i+1)*c(j)
      a(j) = a(j) + b(j,i+2)*c(j)
      a(j) = a(j) + b(j,i+3)*c(j)
   END DO
END DO
```

The references are still all stride one. In each iteration, we need to do one store and six loads. Ignoring cache misses, each iteration takes seven cycles or 1.75 cycles per original iteration. So, we can retain the cache benefits of interchange, while getting a schedule that is almost as good as the non-interchanged case. On the other hand, more registers are required. For this loop, we probably have sufficient registers, but on bigger loops unrolling might lead to register spilling.

From our example, we have seen that we can not make optimal choices about transformations individually. We have also seen that we need to model the entire system in order to select the best overall transformations. We have designed and implemented a compiler algorithm for loop permutation, outer unrolling, tiling, fission and fusion that takes into account all these factors. Essentially, we enumerate the search space of possible transformations, selecting the set of transformations that we estimate will give us the best possible overall performance. By intelligently pruning the search space, we show that we can be both effective and efficient. There is much discussion in the literature on combining transformations to improve cache reuse. For example, Wolf discusses an algorithm combining unimodular transformation (e.g. interchange and skewing) and tiling to quickly determine a good blocking for the cache [W91]. Carr, McKinley, Tseng and Coleman study the order of loops within a tile and methods for choosing block sizes to avoid cache interference [CMT94][CM95]. These works and others provide searching heuristics and models that can be helpful in cache optimization, but they do not consider other aspects of performance. Carr, Kennedy and Guan optimize for cache and instruction scheduling, much as we do [CK94][C96][G95]. However, their work only applies to outer unrolling. The main contribution of this paper is that we combine a wide set of optimizations considering the effects of both caches and instruction scheduling.

In the next section, we briefly discuss the mechanics of our algorithm. Section 3 and 4 are the heart of our paper. Section 3 describes how we estimate the performance of a potential transformation. In Section 4, we discuss how we efficiently search the transformation space to find an effective set of transformations to apply. In Section 5 we compare our results to earlier SGI compilers that used preprocessors from KAI and to Carr and Guan [C96][G95].

2.0 Mechanics of the Transformations

Our algorithm is based on the concept of Singly Nested Loop Nests (SNLNs). A Singly Nested Loop Nest is a perfectly nested loop nest, or an imperfect one where the imperfect code does not contain any loops or other complex control flow. Only *CALL*s, *IF*s and simple statements are allowed in the imperfect code. Consider the example below:

```
DO i = 1, N
   DO j = 1, N
      DO k = 1, N
         IF () ...
         DO l = 1, N
            S1
         END DO
      END DO
      DO m = 1, N
         S2
      END DO
   END DO
END DO
```

This loop contains three SNLNs. One is the nest containing the *i* and *j* loops, one is the nest containing the *k* and *l* loops, and the third is the *m* loop. According to the definition, we could have said that there are five SNLNs: *i*, *j*, *k*, *l* and *m*. We can define *maximal* SNLNs, which result from an SNLN partitioning such that no two SNLNs can be combined to form a single (larger) SNLN. The maximal SNLN partitioning is unique, in this case the one with the three SNLNs.

This definition of maximal SNLN is useful because it is the most general shape of loop nest that we know how to tile, outer unroll and permute. For the purposes of this paper, we do not consider non-inner SNLNs, i.e. maximal SNLNs containing other SNLNs inside. So in the above example, we would not transform the *i* and *j* loops. Our approach can easily be extended to outer SNLNs as well. In fact, since transforming the outer SNLNs does not really affect instruction scheduling, we can treat outer SNLNs separately and only consider how transforming them affects cache behavior.

Let us begin by discussing perfect loop nests. It is a result of standard dependence analysis that if the loops within the nest can be arbitrarily interchanged, then arbitrary til-

ing and arbitrary outer loop unrollings are also legal, as are arbitrary combinations of these transformations. We call such loops *fully permutable*. A Transformable Loop Nest (TLN) is a fully permutable SNLN. To be fully permutable, all dependence arcs from and to array references within the nest must either be carried outside the outermost loop of the nest, or all components of the dependence vectors corresponding to loops within the nest must be non-negative.

Imperfect nests can be reduced to perfect nests in two ways. First, imperfect code might be fissionable, perhaps after application of scalar expansion. Second, we can move the imperfect code into the body of the TLN. For example:

```
DO i = 1, N
   S1
   DO j = i+1, N
      S2
   END DO
END DO
```

can, at least conceptually, be rewritten as

```
DO i = 1, N
   DO j = i+1, N
      IF (j == i+1) S1
      S2
   END DO
END DO
```

Given this rewriting, we can apply dependence analysis to test for legality. Note this rewriting is only legal if we are guaranteed that whenever an iteration of the DO *i* loop executes, then an iteration of the DO *j* loop also executes. We use an integer linear equation solver to prove this legality [MHL91].

The details of transforming non-perfectly nested loops and loops with non-rectangular bounds are beyond the scope of this paper, but it should be known that our system is quite aggressive. We are able to permute any perfectly nested or distributable TLN. We do not interchange other imperfect nests because interchanging them would introduce IFs into the innermost loop. We can tile any TLN, generating efficient IF code (outside the innermost loop) to protect the imperfect computations; we can outer unroll any perfect or distributable nest with invariant bounds, or any TLN of depth 2.

The concept of TLNs leads to a three phase algorithm. In phase 1, we use loop fission and fusion to create deep SNLNs and to collect all maximal SNLNs. Phase 2 transforms the SNLNs and phase 3 reduces register pressure in the innermost loop, if necessary.

Phase 1: Consider the following example,

```
DO i
   DO j1
      DO k1
      END DO
   END DO
   DO j2
      DO k2
      END DO
   END DO
END DO
```

We have the choice of creating one 3-deep SNLN by fusing loop *(j1,k1)* and *(j2,k2)* or creating two 3-deep SNLNs by fissioning loop *i* into two loop nests, *(i1,j1,k1)* and *(i2,j2,k2)*. In the current implementation, we prefer fusion over fission largely because fusion improves cache reuse and causes less loop overhead. Additionally, we have another chance to fission large loops in phase 3 of our algorithm. This is not always the correct decision. If one of the original loops has "bad" dependences, fusing the two loops can inhibit later transformations on the entire nest. If, instead we had chosen fission, we might still have been able to later transform one of the two resultant loops. This situation can be avoided if phase 1 and 2 are more tightly coupled or if we have a back-tracking mechanism. Both are considered as possible future enhancements.

We use a recursive algorithm for phase 1. Given an outer loop, we recursively build maximal SNLNs for all the children of the loop. We then try to use fusion or fission to combine the children SNLNs with the parent loop.

Phase 2: Given the SNLNs built in phase 1, the core of our algorithm searches for the best loop permutation, the best unrolling factors and the best tile sizes. These will be described in detail in Sections 3 and 4.

Phase 3: We mentioned earlier that certain inner loops might not be register allocatable. For each innermost loop that requires spilling, according to the register model described in Section 3.1, we fission the loop into smaller register allocatable loops. The algorithm starts out by building a statement dependence graph in which each node is a statement and the edges indicate scalar or array data dependences among statements. Some of the scalar edges can be removed when scalar expansion is considered. Next we identify the maximal *Strongly Connected Components*, SCCs, in the dependence graph. All statements belonging to the same SCC have to stay in the same loop after fission to preserve data dependences. An SCC dependence graph is then built as a result of an acyclic condensation of the statement dependence graph.

We could fission the loops so that each SCC is in its own loop, but that would be excessive. Our goal is to combine multiple SCCs such that

- as many loops are register allocatable as possible; and
- the number of loops is as small as possible.

We put SCCs without dependence predecessors in a free list. We pick a seed SCC from the free list and then add to the free list those SCCs dependent only on the seed SCC. We select the seed that will result in the most new SCCs being added to the free list. Then we heuristically rank SCCs in the free list. An SCC with more variables in common with the seed SCC will have a higher ranking. Similarly, an SCC containing new variables not in the seed SCC will have a lower ranking. We visit the free list in priority order, and merge into the seed the highest ranking SCC that will not result in register spilling according to the model. This process is repeated until no spill-free merging is possible. Then, a new seed SCC is chosen and the merging process resumes. Finally, we fission the loop and scalar expand those variables with defs and uses in different resultant loops.

Note that our three phase algorithm is not guaranteed to produce the best transformations. It is possible, for example, that in a loop with register allocation problems, it is better to first fission the nest and then try to unroll it. So while we search through the best unrollings, tilings and permutations, we do not do so for each possible fissioning of the loops. We have chosen what we believe is a good compromise that allows us to chose among many interrelated transformations while retaining compile time efficiencies.

3.0 Modeling

Our transformation algorithm depends upon having an evaluation function that can estimate how many cycles a given (possibly transformed) loop nest will take to run on our target. We actually combine the results from two different models. First, the processor model models scheduling and register pressure. Second, the cache model models the cycles lost due to cache misses, and also cycles lost to start-up time of the inner loop, so that the costs of tiling, which often shortens the inner loop, can be accurately computed.

Actually, the models compute cycles per iteration, where in this context "iteration" means a single iteration of the source loop, before transformation. Thus outer-loop unrolling by a factor of two does not approximately double the cycles per iteration, but rather, often decreases it.

3.1 Processor Model

Our goal in this subsection is to estimate how well the code generator will schedule an inner loop given a set of loop optimizations and ignoring cache effects. Without actually scheduling the loop, it is impossible to exactly predict how well a loop will be scheduled. Since we are going to be exploring a large set of potential transformations (potentially hundreds of combinations), we can not generate code, register allocate and schedule each combination. Instead, we use a high level model to estimate bounds on scheduling. Given uncertainty, we bias our model to limit unrolling.

We model three types of constraints: computational resources, latencies and registers.

3.1.1 Computational Resource Modeling

Our machine has a finite amount of resources; FP adders, ALUs, load/store units, etc. We first find a bound on the schedule considering only these resources. We ignore dependences and latencies. We divide the resources into four categories: floating point, memory, integer and issue.

Our instruction scheduler is table driven. Each machine instruction has entries for the resources it will require in different cycles. We use the same table to compute floating point resource limits. We walk the inner loop. For each floating point instruction, we map the instruction into the corresponding machine instruction. For many instructions, this mapping is trivial. A double precision add in the intermediate representation (IR) maps directly into a machine level double precision add. Other instructions are a bit more complicated. For example, a complex add in the IR maps into two floating point adds in the machine level. As another example, a multiply feeding an add in the IR maps into the single "madd" machine instruction. We keep a count for each machine resource. Given a machine instruction, we use the scheduling table to increment the count of each resource used by that instruction. If our machine has multiple units of certain resources, we divide the count appropriately. After walking the code, we find the maximum count over all the resources and use that as our estimate of the floating point resource requirements. Note that this estimate ignores floating point common sub-expressions, or CSEs (common floating point expressions, not common memory references). Since our algorithm runs before the scalar optimizer, we do not know what expressions will be CSE'd. Underestimating the number of CSEs is a conservative error in that it leads to less unrolling.

There is much more commonality among floating point memory references than among floating point operations.

In fact, one of the main reasons we unroll a loop is to expose memory reference CSEs. Therefore, we can not simply walk the code and count the number of references to get an estimate of memory resources. Instead, we walk the code and gather all array references. Scalars need not be loaded or stored inside the loop. Similarly, inner loop invariant array references need not be considered. We put all the variant array references into equivalence classes. Two references are in the same equivalence class if they reference the same location within a small number of iterations, 6,of the inner loop. For example, the two array references a[i] and a[i+1] will be put in the same equivalence class. When referencing a[i+1], our software pipeliner will save the value in a register and will use that value rather than referencing a[i] in the next iteration. If an equivalence class contains both a load and a store to the same location, and the first load comes lexically before the first store, that class requires two memory references. For example, given the statement "a[i] = a[i] ...", we must load the old value of a[i] and store the new value. Otherwise, each equivalence class requires one memory reference. We divide the number of estimated references by the number of memory units to get a memory resource bound.

Numerical codes tend not to have much integer computation. Integer computations are also difficult to estimate at loop nest optimization time. Before scalar optimizations, it is difficult to know how memory addresses will be strength reduced and CSE'd, so we do not know how much arithmetic will be required to compute addresses. We therefore take the conservative approach and assume that every inner loop requires three ALU operations; one branch, one increment to a base register for addressing the array references and one increment for counting loop iterations. We therefore divide three by the number of ALUs. In most loop nests dominated by floating point and memory operations, the number of integer operations do not make a difference. In code well balanced between floating point and memory operations, considering integer operations leads us to unroll loops slightly more than we would have otherwise. Hopefully this exposes more memory CSEs and leads to better schedules.

We consider issue resources separately because they are limited by the combination of other resources. On the R10000, for example, we have resources for two floating point operations, two ALU operations and one memory reference per cycle, but we can only issue four total instructions per cycle. For the issue resource, we add together the number of ALU and memory instructions as computed above. To this sum, we add the number of floating point instructions. The number of floating point instructions is distinct from the floating point resource requirement since, for example, a floating point divide may take up many cycles of resources, but it need only be issued once. Given the sum, we divide by the number of issue units.

Finally, our resource cycle estimate is the maximum of the floating point limit, the memory limit, the integer limit and the issue limit.

3.1.2 Latency Modeling

The cycle time of a loop might also be limited by the latency of dependent operations, in other words the length of a recurrence. Since we are software pipelining inner loops, we do not need to worry about intra-iteration memory dependences. Only inter-iteration dependences affect the latency of a software pipelined schedule. Our algorithm for estimating latencies comes from the software pipelining literature [L88][RGSL96].

We build a dependence graph for the loop. We add a vertex for each floating point load and each floating point store. Each edge in the graph is marked with a latency and a distance. We add an edge to every store from every load in the same statement. These edges will have a latency equal to the sum of the latencies of the operations going from the load up to the store. So for example, given the statement a[i] = (b[i] + c[i]) / d[i], there will be an edge with the latency of a floating point division from the load of d[i] to the store. Similarly, there will be an edge from the load of b[i] and another from the load of c[i] to the store with the latency of a floating point add plus the latency of a floating point division. The reference, d[i], can be loaded immediately before the division while b[i] and c[i] must be loaded before the add, and the add must complete before the division can begin. All the edges will have a distance '0' signifying that the memory references and floating point operations are occurring in the same iteration. Given a flow memory dependence, we add an edge to the graph from the store to the load with latency '0' and the distance in iterations of the dependence. So given, for example

a[i] = ...
... = a[i-2]

we add an edge from the store to the load with latency '0' and distance '2'. This represents the fact that the load in a particular iteration must wait for the store from two iterations earlier to complete.

From the literature, the latency limit of the loop is the maximum over all cycles in the graph of the sum of the latencies in the cycle divided by the sum of the distances in the cycle. We use the algorithm from our software pipeliner to compute this limit.

3.1.3 Modeling Floating Point Registers

Loop transformations can increase register pressure. After some transformations, it can become impossible to schedule a loop without spilling registers.

When modeling register pressure, we consider three effects: pipeline filling, invariant memory references and memory CSEs. Previous approaches have looked at expression trees rather than pipeline filling [CK94][ASU86]. Our approach comes from the belief that on modern superscalar machines, pipeline filling is much more important. Take for example the MIPS R8000 microprocessor. It can perform two floating point multiply-adds, or madds, per cycle, but the latency to load a value, perform a madd and then store the result is five cycles. Therefore, in order to run the machine at peak performance, we need to perform 10 madds in parallel. Each madd must write to a unique result register; otherwise the pipeline will stall. In addition, we need several floating point register as input registers (some of the input registers can be reused before the first madd completes). We define the pipeline filling requirements of a microprocessor to be the number of floating point registers required to keep the pipeline running at full speed. On the R8000, the requirement is 18 floating point registers. On the R10000, which has shorter latencies, the number is 14 registers.

Added to the number of pipeline filling registers are the floating point registers needed to hold the results of memory references. If an array load is invariant, we do not need to load it every iteration, but the price of avoiding a load is one register to hold its value. Similarly, given the two array loads a[i] and a[i-2], the software pipeliner will hold the value a[i] in a register for two iterations in order to avoid the second load. So, we group all the variant memory references in equivalence classes as we did when computing the memory resource requirements. The span of an equivalence class is the distance in iterations between the first and last reference in the equivalence class. The number of registers required to hold all the CSEs is the sum of the spans of all the equivalence classes. We add this number to the number of invariant registers and the number of pipeline filling registers to get the total floating point register requirement.

If the register requirement is less than the number of registers in the microprocessor, then there is no cost to the use of the registers. Otherwise, we can compute the cost as follows. We divide the total number of memory references by the number of registers required to hold the CSEs and invariants. Let's call this ratio r. We assume that we can trade off a register by instead performing r memory references. In other words, rather than keeping an invariant in a register, we can load it from memory, potentially multiple times per iteration. While register spilling can sometimes lead to disastrous results, on codes not limited by memory references we can sometimes spill for free. We add to the number of memory references required r times the number of excess registers required.

We tend to be conservative about unrolling. We never unroll a loop that we estimate will require register spilling, even if we believe that the spilling can be tolerated.

3.2 Cache Model

In our regime, the cache model has two tasks. First, in our algorithm the tile sizes are not known when the model is called, so the cache model must select a good tile size. Second, given that tiling, it determines an estimate for the "cache overhead" and also the "loop overhead"[1].

Suppose, for example, that the compiler needs to model the following matrix multiply loop without permutation and with the i and j loops each outer unrolled by a factor of two. Further suppose that the nest is to be tiled so that a tile of BjxBk is innermost. (The model will be asked to try to evaluate other tilings during other model invocations.)

```
DO i = 1, N
  DO j = 1, N
    c(i,j) = 0
    DO k = 1, N
      c(i,j) = c(i,j) +a(i,k)*b(k,j)
    END DO
  END DO
END DO
```

After the transformations, assuming for simplicity that N and Bj are multiples of two, the example looks as follows.[2]

1. By decreasing the number of iterations in the inner loop, tiling increases loop overhead. In particular, it increases the wind-up and wind-down costs of software pipelining.

2. The compiler does not transform the code until it has determined the transformation to apply. Instead, the model sees the original nest, a description of the current transformation of interest and a description of which variables will need to be scalar expanded.

```
DO j = 1, N
  DO i = 1,N
    c(i,j) = 0
  END DO
END DO
DO jj = 1, N, Bj
  DO kk = 1, N, Bk
    DO i = 1, N, 2
      DO j = jj, MIN(jj+Bj-1,N), 2
       DO k = kk, MIN(kk+Bk-1,N)
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
         c(i+1,j) = c(i+1,j) + a(i+1,k)*b(k,j)
         c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
         c(i+1,j+1) = c(i+1,j+1) + a(i+1,k)*b(k,j+1)
       END DO
      END DO
    END DO
  END DO
END DO
```

The model's tasks are to select a good Bj and Bk and to return a cache and loop overhead cost estimate for the selected Bj and Bk.

In order to achieve its two tasks, the cache model computes a formula for the cost (in cycles per iteration) of this transformation as a function of the unknown tile sizes. It then attempts to select a tile size that minimizes this function.

The model first computes a *footprint*, the number of bytes of the cache used by a reference or set of references within a loop nest. Consider the following example.

```
DO i = 1, N
  DO j = 1, M
    a(i), b(j), c(i,j), c(i+1,j)
  END DO
END DO
```

If each array element is eight bytes, and a cache line is 32 bytes, then our best estimate for the footprint sizes of the arrays are *a*: *(N+3) * 8* bytes, *b*: *(M+3) * 8* bytes and *c*: *M*(N+4) * 8* bytes. The *a* and *b* arrays are each expected to use three extra words because of the long cache lines. The *c* matrix is trickier. The *c* matrix has *M* columns, each of which is just like the *a* array, except that because of the *i* and *i+1*, it is as if there were *N+1* iterations. Note, though, that this estimate is incorrect if *c* is declared *c(N+1,*)*. In that case, there are no wasted cache lines except possibly for the first and last references in the nest. That is, the correct formula for *c* is smaller: *(M*(N+1)+3) * 8*. The exact method we use to compute a footprint is involved and beyond the scope of this paper.

We sort references into uniformly generated sets [GJG88], which are sets of references to the same array whose index expressions differ by at most constants in each dimension. We then compute one footprint for each uniformly generated set. That means that we compute a single footprint for the references *a(i,j)*, *a(i,j+1)* and *a(i+1,j)*. To compute the entire footprint for many references, we simply add the footprint for each uniformly generated set. This is logical, since there is likely to be very little reuse between uniformly generated sets. Thus, we add the footprints for *a(i)* and *b(i)*, and even for *a(i)* and *a(j)*.

We compute a total footprint for each subnest. In our matrix multiplication example above, we only need to evaluate footprints for the following nest:

```
DO i = 1, N, 2
  DO j = jj, MIN(jj+Bj-1,N), 2
    DO k = kk, MIN(kk+Bk-1,N)
```

There is no reuse across the *jj* and the *kk* loops as we chose the block sizes to be sufficiently large so that the inner tiles use up all of the effective cache. We compute a footprint for the *k* loop (the *Bk* iterations of *k*), a footprint for the *j* loop (the *Bj*Bk* iterations of the nest rooted at *j*), and a footprint for the *i* loop (the *N/2*Bj*Bk* iterations of the nest rooted at *i*). At this point, *Bj* and *Bk* are unknown; we develop a formula in which *Bj* and *Bk* are variables.

We say that loop *i* overflows the cache if the sum of the footprints for that loop exceed the cache size. Since most caches are not highly associative, we don't use the actual cache size, but instead the *effective cache size*. Since we do not attempt to model actual cache line interference[1], we must estimate how much of the cache we can use. For example, one study [LRW91] showed that using blocks that used only from 5% to 15% of the cache was optimal when we did not use more precise information about cache mappings. If we have more highly associative caches, smaller caches or shorter cache lines, the optimal percentage is larger. For each cache in the memory hierarchy, we choose a value based upon its characteristics and call that the effective cache size.

If the entire three-deep loop nest without tiling (i.e., with *Bj=Bj=N*) does not overflow the cache, then tiling only adds loop overhead and is harmful, so we do not tile.

In our example nest, *N* is unknown. Therefore, we assume that the *i* loop does overflow the cache, and we probably

1. Our model has a special provision for the case when array dimensions are near powers of two, since cache interference is a particular risk at that point, but that's beyond the scope of this paper. Cache line interference can be modelled more aggressively using techniques from [CM95].

want to tile, assuming we find sufficient reuse in the i and j loops to justify the overhead in the k loop. Ignoring edge effects to simplify the presentation, for this loop, the footprint for the i loop given that j is blocked by Bj and k by Bk is approximately $Fi = 8*(N*Bk + Bj*Bk + N*Bj)$ and likewise $Fj = 8*(Bk + Bj*Bk + Bj)$ and $Fk = 8*(Bk + Bk + 1)$. For simplicity of presentation, we will discuss the simplest version of the model in which Fk does not play a role, and we will ignore the fact that read misses and write misses might affect performance differently.

Our model assumes that in the first iteration of i, Fj bytes are brought in. Thus, on each subsequent iteration of i, roughly $(Fi-Fj)/(N-1)$ bytes are brought in. At this point, we guess that $N=100$; the exact value does not greatly effect the model.

Let d be the number of iterations of i across which data must live for all reuse to be taken advantage of. For example, we have $c(i,j)$, so to use the cache line maximally, data must survive in the cache through two iterations of i, since the cache line size is four, but we are stepping by two. The amount of the cache required to take advantage of all the reuse, the *requirements*, is approximately $Fj + (d-1)*(Fi-Fj)/(N-1)$. Assuming no cache interference, which in our model means that the requirements are much smaller than the effective cache size, the total number of misses will equal the *requirements* * N/d. When the requirements are close to the effective cache size, the interference is pretty small but not non-existent. We model the percent of the footprint that are misses because of cache interference to be proportional to *requirements/effective_cache_size*. Once we exceed the effective cache size, we add in another similar but much larger term proportional to *(requirements-effective cache size) /effective cache size*.

To convert the number of misses into a cycles per iteration penalty is not a matter of simply multiplying the miss rate by the miss penalty. Depending on cache characteristics (multiple outstanding references, prefetching, etc.), some of the cache miss cycles may be hidable by the processor cycles. So the cache model is passed the estimate of the processor cycles from the Processor Model (see Section 3.1) and uses this number to estimate how many of the cache miss cycles will actually stall the machine. The details are beyond the scope of this paper.

We add in our estimate of loop overhead, which is roughly $(((o/N) + o)/Bj + o)/Bk$ per iteration, where o is a constant appropriate for this processor and loop nest. This term will cause the compiler to prefer rectangular blocks when choosing Bj and Bk, with the inner loop executing more iterations, as is appropriate. Finally, we add in a term for the TLB, which is like small cache with a huge cache line. In Section 4, we describe how we use our formula to find the ideal block sizes.

4.0 Searching for the Best Transformation

FIGURE 1. High-level description of TLN transformation algorithm

```
best_cost = infinity
best_trans = none
FOR each possible innermost loop DO
  unroll_trans = compute best outer
      unrolls for this inner loop
  unroll_cost = compute cost for this unrolling
  tile_trans = compute best outer permutation
      and tiling given unroll_trans
  tile_cost = compute cache and loop overhead
      cost given unroll_trans and tile_trans
  IF (tile_cost + unroll_cost < best_cost)
      best_cost = tile_cost +unroll_cost
      best_trans = unroll_trans and tile_trans
  END IF
END FOR
```

In this section, we show how we search through the possible transformations to find the set of transformations that lead to the best performance according to the models developed in the previous section. The number of possible different permutations, tilings and outer unrolls, and the distributions and scalar expansions that enable them is too large to search exhaustively. We prune the search by recognizing that there are different combinations of transformations that should have very similar performance and by evaluating combinations that might reasonably differ from each other in expected performance. Figure 1 contains a high-level description of our TLN optimization algorithm. This algorithm involves a major pruning of the search space, since decisions about outer unrolling are made independently from any tiling or permutation of non-innermost loops decisions. We can understand why this approach is effective by understanding how the different transformations alter scheduling, cache overhead and loop overhead, individually and in combination.

For scheduling, only the choice of innermost loop and the choice of outer unrolling factors matter. For cache blocking, the dominant factors are which loops are tiled innermost and, to a lesser but still important extent, the order of the loops within the tile. Outer unrolling does affect cache performance, but tiling is more effective than outer unrolling for cache optimization since it allows for larger tile

sizes. Thus, only the choice of innermost loop is important to both, and our algorithm simply enumerates all possible innermost loops. Given each innermost loop, we can select the transformations to improve scheduling and the transformation to improve the cache relatively independently. Since outer unrolling affects the cache more than cache transformations affect the schedule, we choose the unrolling first and feed that into the cache model.

4.1 Selecting the Scheduling Transformation Given a Potential Innermost Loop

FIGURE 2. Choosing Outer Unrolling Factors

```
MAXPROD = 16 ; MAXUNROLLS = 16

algorithm select_driver(num_loops, inner_loop,
                        best_unrolls, best_cost) {
  Unrolls = Best_Unrolls = {1, 1, ...}
    Best_Cost = infinite
  Found_Ideal = FALSE

  if (num_loops >= 2) {
    can_reg_allocate = TRUE
    call select(1, 0, num_loops, inner_loop,
                        &can_reg_allocate)
  }
}
procedure select(curprod, curloop, num_loops,
                    inner_loop,can_reg_allocate) {
  BOOL can_allocate = TRUE;
  for u = 1 to MAXUNROLLS {
    if (u*curprod > MAXPROD || Found_Ideal ||
                    ! can_allocate) break
    Unrolls[curloop] = u;
    if (curloop < num_loops-1) {
      select(curprod*u, curloop+1, num_loops,
                    inner_loop, &can_allocate)
      if (u == 1 && can_allocate)
        *can_reg_allocate = FALSE;
    } else {
      this_cost = evaluate(Unrolls, inner_loop,
                                    &can_allocate)
      if (this_cost < best_cost) {
        Best_Cost = this_cost
        Best_Unrolls = Unrolls
      }
    }
    if (curloop == inner_loop) break; // we do not
                        //unroll the inner loop
  }
}
```

Given a potential inner loop, we use the algorithm in Figure 2 along with the processor model in Section 3.1 to decide how much to unroll each outer loop. Essentially we enumerate all possible combinations of unrolling factors for all the outer loops. In contrast, Carr and Kennedy chose unrolling factors by solving an optimization problem in terms of the original operations, references and data dependences[CCK88] . This allows them to potentially be more efficient but at a small loss of accuracy. By enumerating the possible unrolling factors, we can potentially be completely accurate.

We limit ourselves to combinations where the product of all the unrolling factors is no larger than 16. We have not found a case in practice where larger factors can help. To improve compiler efficiency, we stop evaluating combinations whenever we find an "ideal" schedule. An "ideal" schedule is one limited by floating point resources. Unrolling further can not improve the floating point resource bound. We also do not evaluate combinations that we know we will not be able to register allocate. Consider for example a three deep nest. If we discover that the combination (3,1,1) (unrolling the outer loop by three) is unallocatable, we will not try the combination (4,1,1) nor the combination (3,2,1) since they are guaranteed to be unallocatable. We will, however, try the combination (2,2,1), since it may be allocatable.

We do not generate code for each combination we try; that would be much too inefficient. We do not perform any transformations to the code until we decide on which combination is best. Instead, we designed our evaluation algorithm to come up with a schedule estimate in terms of the original code and a potential transformation. Let us consider the various components of the estimate. The floating point resource bound is not affected by unrolling. The number of cycles required for the transformed source increases with the unrolling product, but since we compute an estimate per original iteration, i.e., normalized by the unrolling product, the floating point resource estimate is constant. Therefore, we only compute it once for a given loop nest. The memory resource bound is affected by unrolling. So, we must compute it for each possible combination. But, we do not unroll the code in order to compute it. Instead, we unroll a compact data structure describing all the array references. We then compute CSEs and invariants directly on this data structure. From Callahan, Cocke and Kennedy [CCK88], the latency limit per original iteration goes down directly in proportion to the unrolling product. So, we only compute the latency limit once per choice of potential inner loop. The pipeline filling register requirement is a constant for a given target architecture. The register requirements for invariant references and CSEs need to be evaluated once per unrolling combi-

nation. Once again, we unroll our data structure rather than the code.

We limit our transformations to nests of depth eight or less. If a nest is more deeply nested, we do not transform the outermost loops. Assuming a depth of eight, in the worst case there are 3,425 unrolling combinations for each potential inner loop. As we show in Section 5, in practice we are very efficient.

4.2 Selecting the Cache Transformation

FIGURE 3. Algorithm for Choosing Tiling for a Single Cache

```
tile_cost = infinite
tile_trans = null
Select locality_loops as follows:
   only select loops with some reuse.
   if more than three have some reuse, pick three with
        most reuse (loops with most reuse are precom-
        puted once for the nest)
FOR loops in POWER_SET(locality_loops)
   FOR each possible permutation of loops
        this_tile_trans = compute best tile sizes for this
             tiling/unrolling
        this_tile_cost = compute cache and loop
             overhead cost for this tiling/unrolling
        IF (this_tile_cost < tile_cost)
          tile_cost = this_tile_cost
          tile_trans = this_tile_trans
        END IF
   END FOR
END FOR
```

Figure 3 shows our algorithm for selecting a permutation and tiling for a cache. It begins by selecting all the non-innermost loops that have reuse. If there are more than three, then only the three with the most reuse are selected[1]. We do this pruning to keep the algorithm fast. Since it will not help cache performance to include a loop that has no reuse in the tiling, this pruning only may hurt when more than three loops, not including the innermost loop, have reuse. This case is very rare and, since we still tile the heuristically best loops, not at all catastrophic when it occurs.

We exhaustively select for tiling none, some or all of the (pruned) loops that have locality. We look at all possible orderings of those loops. For example, suppose we have the following nest

```
DO i = 1,N
  DO j = 1,N
    DO k = 1,N
```

and we are exploring loop k innermost, and both i and j have reuse. Then we will explore the following possibilities:

```
loops = {} -> (1) DO i = 1,N
                  DO j = 1,N
                  DO k = 1,N
loops = {i} -> (2) DO j = 1,N
                  DO kk = 1,N,Bk
                   DO i = 1,N
                    DO k = kk, MIN(N,kk+Bk-1)
loops = {j} -> (3) DO i = 1,N
                  DO kk = 1,N,Bk
                   DO j = 1,N
                    DO k = kk, MIN(N,kk+Bk-1)
loops = {i,j} -> (4) DO jj = 1,N,Bj
                    DO kk = 1,N,Bk
                     DO i = 1,N
                       DO j = jj, MIN(N,jj+Bj-1)
                       DO k = kk, MIN(N,kk+Bk-1)
                (5) DO ii = 1,N,Bi
                   DO kk = 1,N,Bk
                    DO j = 1,N
                     DO i = ii, MIN(N,ii+Bi-1)
                       DO k = kk, MIN(N,kk+Bk-1)
```

Notice that the permutation determines the order of the inner loops. We tile all those loops except the outermost of them, and move those outer tiles outside the untiled loop. There are of course other ways to tile. For example, when loops = {j} above, we could instead have generated

```
DO i = 1,N
    DO jj = 1, N, Bj
      DO kk = 1, N, Bk
        DO j = jj, MIN(N,jj+Bj-1)
         DO k = kk, MIN(N,kk+Bk-1)
```

However, there is typically no advantage to this code over version (3) for a uniprocessor with a one-level cache. If $Bj = N$, then we take advantage of reuse in Bk iterations of k and N iterations of j. If $Bj < N$, then we lose some reuse in the j loop. Can we use a larger Bk to make up for it? No, because we could then have used the same Bk in (3) without ill effect. But with a small Bj, might we avoid overflowing the cache, thereby getting reuse in loop kk? When there's a great deal of reuse in k and very little reuse in j (but not so little that we don't want a few iterations in the tile), this can happen. So the pruning we have done by not

1. For each loop, we compute the footprint per iteration assuming that loop to be innermost, a technique inspired by [CMT94].

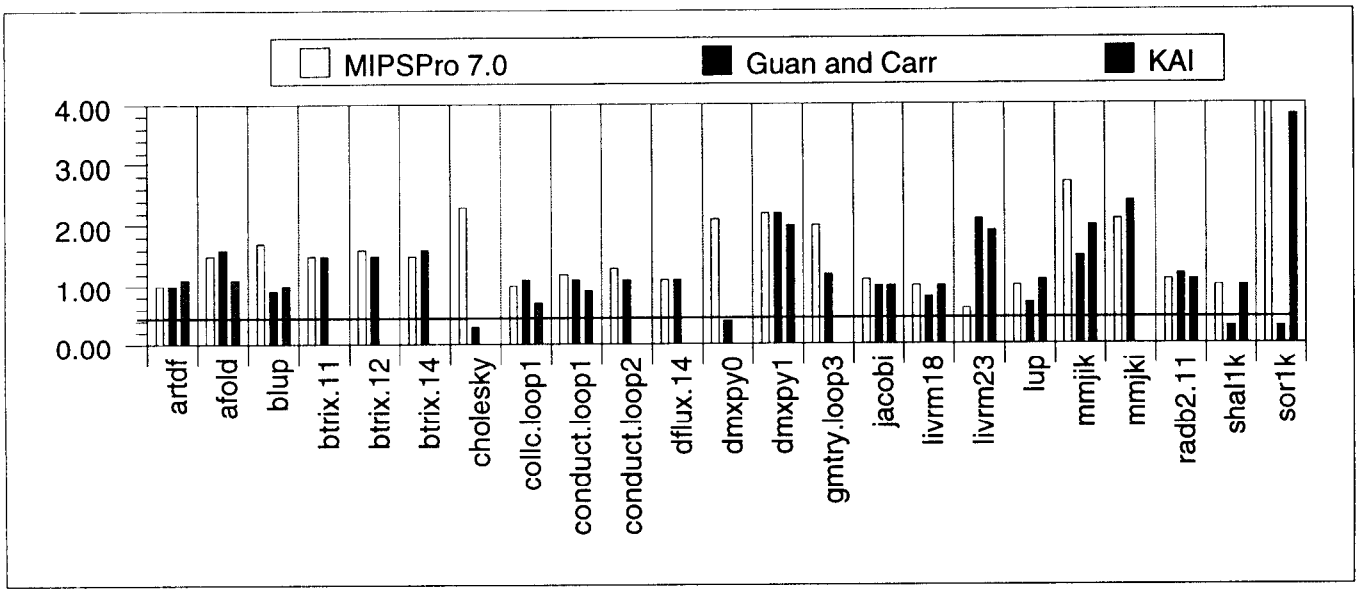

Figure 4: Outer Loop Unrolling for the R8000

considering this case can on very rare occasions be suboptimal.

Given the choice of which loops to tile, we need to choose the appropriate tile sizes. All we know about the tile sizes Bi is that they are positive, integral (multiples of the corresponding unrolling factors) and may be bounded (if we know the loop bounds). We pass these constraints to a solver. It does a (recursive) binary search for the best values. This works perfectly so long as the formula has only one minimum. Otherwise it might come up with some local minimum. The formulas we develop in fact produce very good answers in practice. The solver produces tile sizes and a cycles per iteration estimate. The run time of this search is $O((logN)^n)$, where n is the number of unknowns and N is the loop bound size. If N is unknown, then we heuristically use 100,000 as a bound. This search is fast, although to make sure it remains fast for deep loop nests, we require that all tile size variables be the same except for the innermost tile size. So, if we tile *Bi*, *Bj* and *Bk*, with *Bk* innermost, we require that *Bi=Bj*.

With these restrictions, we have reduced the search space significantly. It is easy to show that the number of times the inner loop of our algorithm is executed is $\Sigma_{i=0,n}\binom{n}{i} \times i!$, which is exponential in n. However, we have bounded n at three, so that the inner loop may execute at most 16 times, and then only when three loops other than the innermost loop have locality. The inner loop itself is not a function of n. So the $n=3$ case (four loops in the TLN) takes 32 times as long as the $n=1$ case (two loops in the TLN), but the algorithm does not slow down after that.

Note that we could instead execute the innermost loop $n+1$ times (again bounding n by 3 if we wish) by permuting the outer loops to execute in memory order, and just tiling the innermost loop and up to n outer loops, a la [CMT94]. We have found this inadequate because, in practice, the one number computed in the memory order computation does not adequately describe the situation. If loop bounds are small on a loop with good locality, the added overhead may or may not be a problem, depending on the tile sizes of innermore loops. Also, the untiled loop gets many more iterations but also has a greater risk of having reused data being removed from the cache; we give the model the chance to trade off these factors, possibly choosing small tile sizes further in, if overhead permits, to try to gain more reuse outside. Because of these situations, we have rejected the memory order approach, in favor of a slower but more accurate algorithm.

5.0 Experimental Results

Our algorithm is implemented as part of the MIPSPro 7.0 compiler, the new production compiler for SGI machines. We tested our algorithm using a set of kernels supplied to us by Guan [G95] and Carr. These are kernels extracted by them from SPEC, Perfect and RICEPS. Guan and Carr use an algorithm similar to ours for outer loop unrolling, but they do not consider interchange, cache blocking, fission nor fusion. To compare ourselves to them, we turned off these optimizations in our compiler. Since we were trying to isolate the non-cache effects of unrolling, we used a MIPS R8000 system, a 300MFlops peak computer, with a large, 4MB primary cache. We adjusted the size of a few of the benchmarks to make sure they fit inside the cache.

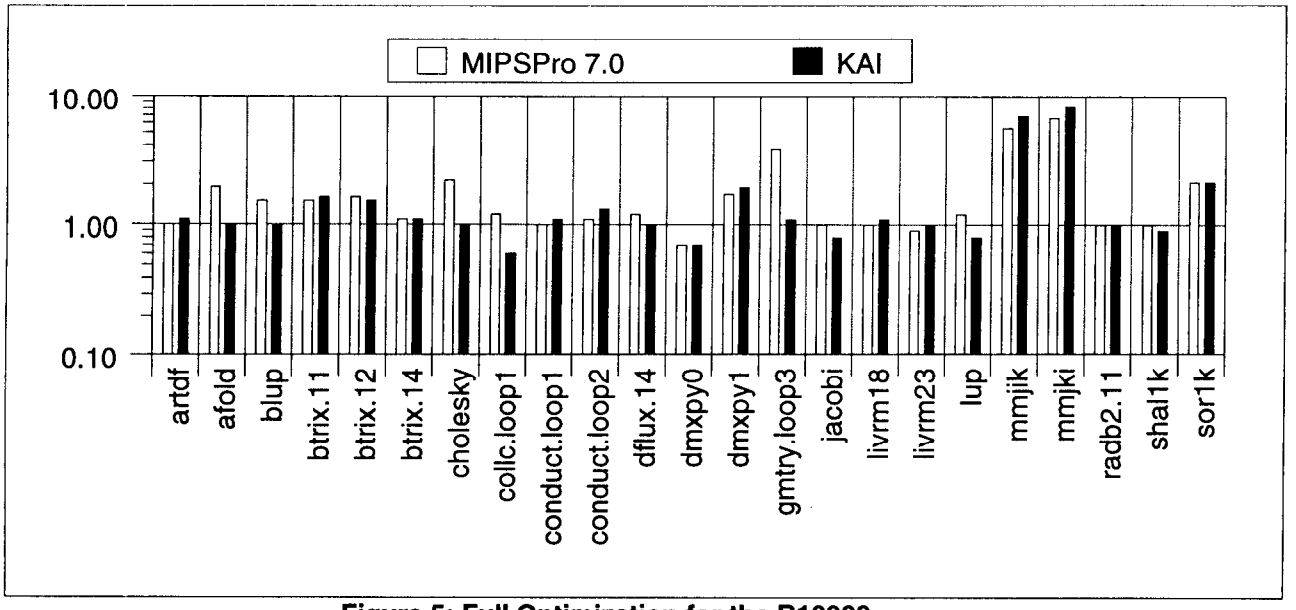

Figure 5: Full Optimization for the R10000

Figure 4 shows our results compared to Guan and Carr and compared to an earlier version of the MIPSPro compiler that used a preprocessor from Kuck and Associates. We normalized all results against a run with our compiler with outer loop unrolling turned off. Numbers larger than one indicate that unrolling improved performance. Since we were not able to turn off interchanging in the KAI preprocessor, we do not include numbers for KAI for those benchmarks that KAI decided to interchange. For all three systems, we used the same compiler back end from the new MIPSPro compiler. Thus our comparisons should only be affected by unrolling. A few of the kernels did not come with main programs. In one of the kernels, Guan and Carr unroll a timing loop. We do not include these kernels.

Our results look quite promising. Our only poor result is livrm23. In that case our unrolling confuses our software pipeliner due to a performance bug in the software pipeliner. By judiciously adding a set of parenthesis, we still unroll the same amount, but we eliminate the confusion and get good results. Guan and Carr do well on many kernels, but they do very poorly on a few. At the time we generated results, their algorithm was not well tuned for the R8000, and they overunrolled a few kernels, causing serious register problems. We believe that they would do better now. Our algorithm, though, has two advantages. Using the pipeline filling concept to estimate register pressure allows us to optimize for multiple machines without having to tune for each machine specifically. Second, because Guan and Carr use unrolling to improve cache behavior, they have a tendency to unroll more than we do. We use cache tiling, rather than unrolling to improve cache behavior. Cache tiling is more effective than unrolling in reducing cache misses, and it does not increase register pressure. In kernels suffering from large register requirements, if they model register pressure perfectly they will limit unrolling and therefore lose some of the cache benefits. If registers are not perfectly modeled, there is a larger probability that they will overunroll than that we will.

KAI is less successful than the other two algorithms on several kernels. On several kernels they either do not unroll or do not unroll sufficiently.

As a second comparison, we turned on all our optimizations and compared ourselves to KAI. To place more emphasis on cache effects, we ran on a newer machine, a MIPS R10000, with a smaller cache. In addition, we increased the data size of some of the kernels so that they would exceed the cache size of the R10000. Once again, we used the same back end for both our algorithm and KAI. The results are given in Figure 5.

Our optimizations improve performance on average (geometric mean) by 50%. KAI's improve performance by 30%. The only example that we do very poorly on is dmxpy0, a matrix-vector product. We perform an interchange that confuses our prefetching algorithm, so that we do not get any benefit from prefetching while the base algorithm gets a large benefit from prefetching. Part of the problem can be considered a performance bug in the prefetching algorithm, but part of the problem comes from the fact that we do not adequately integrate the prefetching algorithm into the rest of our decision algorithm.

Our algorithm may appear to be expensive. We are potentially enumerating a very large search space. We timed our compiler on an R10000 system compiling the SPECfp95 benchmarks. These are a set of fortran programs ranging

from 200 to 7,000 lines. The total compilation times ranged from 2.9 to 74.2 seconds. The amount of time spent modeling both the processor and the cache ranged from 0.0 to 4.0 seconds. The amount of time spent both modeling and actually performing the optimizations mentioned in this paper ranged from 0.1 to 8.8 seconds. In the worst case, 22% of the compile time, or 1.2 out of 5.9 seconds, was spent modeling and performing the optimizations.

6.0 Conclusion

We have presented a compiler algorithm that applies fission, fusion, tiling, permutation and outer loop unrolling to optimize loop nests. This algorithm is based upon a model that estimates total machine cycle time taking into account cache misses, software pipelining, register pressure and loop overhead. The algorithm intelligently searches through the various possible transformations, using our model to select the set of transformations leading to the best overall performance. Our algorithm is part of a production compiler system. We have tested it and found it to generate, with a few exceptions, better code than two other systems. While our algorithm may appear to be inefficient, we have found its compile time to be very reasonable

ACKNOWLEDGEMENTS

We would like to thank Yiping Guan and Steve Carr for their generous help in providing us their benchmarks and the output of their compiler on the benchmarks.

Bibliography

[ASU86]Aho, A., Sethi, R., and Ullman, J. Compilers, Principles, Techniques, and Tools. Addison-Wesley, 1986.

[CCK88]Callahan, D., Cocke, J., and Kennedy, K. Estimating interlock and improving balance for pipelined machines. Journal of Parallel and Distributed Computing 5, 334-358, 1988

[C96]Carr, S.. "Combining Optimization for Cache and Instruction-Level Parallelism", In Proceedings of the 1996 Conference on Parallel Architectures and Compiler Techniques, Boston, MA, October 20-23, 1996.

[CK94]Carr, S. and Kennedy, K. Improving the Ratio of Memory Operations in Floating-Point Operations in Loops. ACM Transactions on Programming Languages and Systems 16(6) 1768-1810, Nov. 1994.

[CMT94]Carr, S., McKinley, K.S., and Tseng, C.. Compiler Optimizations for Improving Data Locality. Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

[CM95]Coleman, S. and McKinley, K.S., Tile Size Selection Using Cache Organization and Data Layout, In SIGPLAN'95: Conference on Programming Language Design and Implementation, June 1995.

[GJG88]Gannon, D., Jalby, W., and Gallivan, K. Strategies for cache and local memory management by global program transformation. Journal of Parallel and Distributed Computing, 5:587-616, 1988.

[G95]Guan, Y.. Unroll-And-Jam Guided by A Linear-Algebra-Based Data-Reuse Model. Masters Thesis, Computer Science Department, Michigan Technological University, 1995

[L88]Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. In Proceedings of the '88 SIGPLAN Conference on Programming Language Design and Implementation, June 1988.

[LRW91] Lam, M., Rothberg, E., and Wolf, M.E. The Cache Performance and Optimizations of Blocked Algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.

[MHL91]Maydan, D., Hennessy, J, and Lam, M., Efficient and Exact Data Dependence Analysis, In SIGPLAN'91: Conference on Programming Language Design and Implementation, June 1991

[RGSL96]Ruttenberg, J., Gao, G.R., Stoutchinin, A., and Lichtenstein, W., Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler, In SIGPLAN'96: Conference on Programming Language Design and Implementation, June 1996

[W91]Wolf, M.E., A Loop Transformation Theory and Improving Locality and Parallelism in Nested Loops, Ph.D. thesis, Stanford University, August 1992.