Cost Effective Memory Disambiguation for Multimedia Codes

Esther Salamí, Jesús Corbal, Carlos Álvarez, Mateo Valero *
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
{esalami, jcorbal, calvarez, mateo}@ac.upc.es

ABSTRACT

Frequently, ambiguous memory references prevent the compiler from exploiting all the available ILP. Techniques to detect aliasing between access patterns of array elements are quite effective for many numeric applications, but although media codes usually process disjointed streams that exhibit regular access patterns, current commercial compilers remain unsuccessful in disambiguating them due mainly to complex pointer references. In this paper we propose a very cost effective disambiguation method that takes advantage of the specific behavior of typical media memory patterns. The compiler generates two versions of the same loop and a simple test block that decides at run-time whether or not the entire loop is disambiguated. No additional hardware is required and the increase in compilation time and code size is minimal. We have introduced this technique in Trimaran and evaluated it for a VLIW architecture with guarded execution. Experimental results confirm significant speedups (up to 1.32X for a 4-way architecture) for a relevant percentage of applications from the Mediabench benchmark suite. Furthermore, performance scales up to a 16-way machine (up to 1.73X versus the 16-way baseline).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Languages

Keywords

Multimedia, VLIW, memory disambiguation, run-time analysis, time-to-market

*This work has been supported by the Ministry of Science and Technology of Spain under contract CICYT TIC2001-0995-C02-01 and by the CEPBA.

CASES 2002, October 8–11, 2002, Grenoble, France.

1. INTRODUCTION

To make effective use of VLIW processors, the compiler must be able to exploit Instruction-Level Parallelism (ILP). However, ambiguous memory dependences often limit the compiler ability to perform code optimizations. If there is any possibility that two memory operations reference the same memory location, the compiler must place dependence arcs between them to ensure that they are executed in sequential order. Both static and dynamic memory disambiguation approaches have been proposed in the literature to determine if dependence actually exists for a pair of ambiguous memory references.

Static dependence analysis attempts to solve the ambiguity at compile time. A lot of work has been done to deal with multidimensional arrays and complex array subscripts[8, 14, 18]. However, these techniques are quite ineffective when the access pattern is non-linear or when some essential information, such as loop bounds, is not known at compile-time. Pointer referencing is also one of the most important handicaps to dependence analysis. Although interprocedural analysis techniques have been proposed that provide good pointer disambiguation[4, 25, 6, 27, 3], specially for pointer-intensive applications such as those of SPECint, they result commonly on increases in compilation time and memory requirements.

Dynamic memory disambiguation determines at run-time whether two memory instructions ever reference the same location. The compiler provides different execution paths, and at run time it is determined which one must be followed depending on the existence or not of the dependence[17, 7, 9]. Most of these approaches focus on loops whose memory operations are independent except perhaps for a few iterations. They obtain the gain of a good scheduling at the cost of a penalty on those iterations in which the dependence really exists. A different point of view is to consider the problem of simply deciding if the loop is fully disambiguated or not, that is, determining whether or not there is a dependence in any iteration[2]. The Privatizing DoAll Test [22], for instance, identifies fully parallel loops at run-time and dynamically privatizes scalars and arrays; they obtained significant speed-ups on Fortran loops running on multiprocessor architectures. Other sophisticated approaches exist that produce predicates that may be used either at compile time or at run-time depending on whether there is enough information available[16, 20].

Multimedia applications share different treats with both numerical applications (such as those in SPECfp) and integer applications (such as those in SPECint). As in numer-

```
for(i=0; i<N; i++)                  for(i=0; i<N; i++)
   A[i] = f (B[i], C[i], ... );        A[i] = f (A[i], ... );

              (a)                                 (b)

for(i=1; i<N; i++)                  for(i=0; i<N; i++)
   A[i] = f (A[i-1], ... );            s += f (A[i], B[i], ... );

              (c)                                 (d)
```

Figure 1: Typical multimedia loops

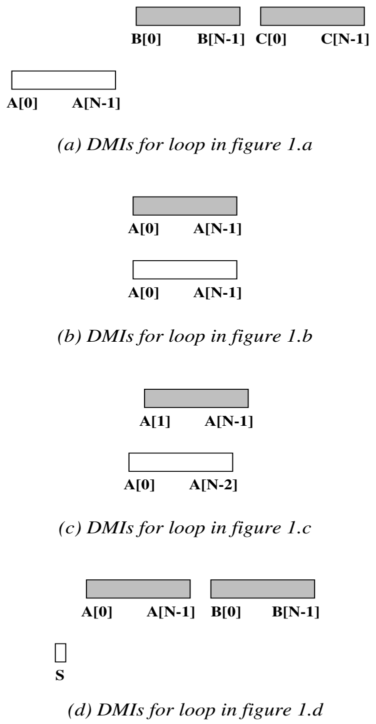
ical applications, media programs make extensive use of n-dimensional data structures with relatively simple patterns. As in integer applications, media applications make extensive use of pointers (since C and C++ are the languages of choice of media developers), sometimes with several levels of indirection to match the media structures of standardized protocols. At the same time, media applications differ from these two wide fields in the characteristics of the data processing. Media applications are streaming; that is, even if there are hard-to-deal control and data dependences in the computation, typical media kernels process one or more input streams of data to provide one output stream of data. Moreover, the input and output streams are typically disjointed regions.

The main focus of this paper is on presenting a very fast to implement memory disambiguation technique that makes sense in the context of media applications, or other kind of programs where input and output data streams are usually disjointed. Based on the characteristics of media applications, we propose a dynamic alias analysis method based on disambiguating at the loop level rather than at the iteration level. Disambiguation can be easily determined by dinamically analyzing the region domain of every load and store stream. Significant gains are obtained at nearly no cost and without the inherent complexity of interprocedural analysis. Section 2 justifies and describes the proposed memory disambiguation method. Section 3 explains the main compilation aspects about the implementation of the algorithm and section 4 presents some experimental results. Finally, section 5 concludes the paper.

2. MEMORY DISAMBIGUATION AND MULTIMEDIA

2.1 Rationale for Cost Effective Disambiguation Based on Dynamic Memory Intervals

Array references in multimedia applications use to follow strided and very simple access patterns. Figure 1 summarizes the kind of loops more commonly found in these codes. The loop in (a) operates over one or several streams to produce another one. If there is no aliasing between the vectors, different array elements are accessed on each iteration, and thus no memory dependence exists. Nevertheless, when these arrays are accessed through pointers, as is usual in media codes, an accurate interprocedural analysis is required to ensure that no aliasing occurs. Such techniques are not generally included in common commercial compilers, so they must be conservative and place dependence arcs between the memory operations in order to ensure correctness. *Output dependences* (dependences between two stores) and *anti dependences* (when a load precedes a dependent store) usually



(a) DMIs for loop in figure 1.a

(b) DMIs for loop in figure 1.b

(c) DMIs for loop in figure 1.c

(d) DMIs for loop in figure 1.d

Figure 2: Dynamic Memory Intervals representation for loops in figure 1

have little impact on the generated code, but *flow dependences* (when a store precedes a dependent load) tend to be a sever restriction for the compiler. In the example, the potential cross-iteration flow dependences from the store of A in the iteration i to the loads B and C in the iteration $i+1$ restrict aggressive techniques, such as modulo scheduling [21], resulting on a poor code scheduling.

In loop (b), the input and output streams coincide. In that case, cross-iteration dependences do not exist either, as loads from iteration $i+1$ do never refer the same memory location as stores from iteration i. The opposite case is shown in (c), where there is a recurrence with distance one, and thus, loads from $i+1$ must not be allowed to advance the stores from i. The last case (d) shows a loop that operates array elements and accumulates the result on a scalar variable S. A register will probably be assigned for the scalar and dependences between two loads (*input dependences*) are not a problem, so there are not ambiguous memory dependences obstructing the compiler in that case.

In contrast with numerical applications, where the identification of the array elements accessed by a particular reference becomes really important for compiler optimizations, we observe that memory references on media loops use to be always dependent or non dependent at all, based on the overlapping or no overlapping of the memory space traveled by the arrays. In other words, we have found that in multimedia we have two main kinds of stream behavior: one where all the input and output streams are totally independent, and other one where the streams have recurrences between themselves. A cost-effective approach to perform memory disambiguation would just need to determine, for every loop, which case we are facing. We will demonstrate

	Mod.Sched. Loops		Loops w.potential	
benchmark	#loops	%instr.	#loops	%instr.
cjpeg	5	74.25%	4	64.53%
djpeg	3	71.93%	3	71.93%
mpeg2enc	1	0.61%	1	0.61%
mpeg2dec	3	19.15%	3	19.15%
toast	4	54.98%	2	5.88%
untoast	2	4.32%	2	4.32%
epic	2	50.45%	1	1.91%
unepic	5	28.37%	3	20.97%

Table 1: Coverage

that such a technique reaches a performance very similar to the idealistic case.

We define the *Dynamic Memory Interval* (DMI) of a memory operation as the memory space limited by the lower and upper locations referred by the operation across all iterations during one invocation of the loop. Figure 2 shows the dynamic memory interval representation for the memory operations of the previous loops. Note that the scalar reference is also included, as it can be considered a zero strided array. If we are able to prove before entering a loop that the dynamic memory intervals of two references do not overlap, we can ensure that they are not memory dependent in that invocation.

2.2 Multimedia Loops Characterization

This section presents some preliminary studies to demonstrate the potential of our proposal.

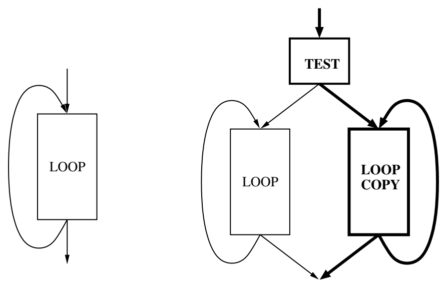
2.2.1 *Benchmark Suite*

We focus on eight representative programs of image, audio and video, all from the UCLA *Mediabench* suite [12]:
- `cjpeg` and `djpeg`: JPEG compression and decompression standard for either gray-scale and color digital images.
- `mpeg2enc` and `mpeg2dec`: encoder and decoder of the MPEG2 standard for high-quality digital video transmisssion.
- `toast` and `untoast`: encoder and decoder of the european GSM 6.10 standard for full-rate speech transcoding.
- `epic` and `unepic`: image compression and decompression.

2.2.2 *Coverage*

Solving ambiguous memory dependences becomes specially profitable to software pipelining techniques such as modulo scheduling. Table 1 shows the number of loops that can be modulo scheduled and the percentage of dynamic instructions they represent (only loops that suppose more than 0.75% of the program instructions are included). But not all have the potential to be disambiguated, as is the case of loops without store operations. The last two columns of the table reflect the number of loops and percentage of dynamic instructions for loops with potential to be disambiguated.

We observe that only three of the eight benchmarks show a low coverage. The rest of the benchmarks present a reasonable number of loops to optimize to grant good performance improvements as a result of including memory disambiguation. Specially interesting are `cjpeg` and `djpeg`, which show a really high potential of improvement based on coverage issues. It is important to note that our technique adds near-zero overhead over those codes that could not benefit from memory disambiguation, thus, overcoming the fact that there exist benchmarks without room for improvement.



(a) Original control flow *(b) New control flow*

Figure 4: Loop duplication

2.2.3 *Potential Gains*

To demonstrate the importance of memory disambiguation on multimedia applications, our benchmark suite was evaluated using oracle memory disambiguation. To model this ideal disambiguation, a list of all independent memory references was first obtained by profiling; afterwards, the program was compiled assuming that all memory references in the list were independent. Figure 3 shows the speed-ups obtained for different way VLIW architectures with and without oracle memory disambiguation.

Results from the figure show that memory disambiguation is a key technique to allow an effective exploitation of the available ILP when we increase the way of the machine. The oracle version of the code provides a 17.23% of improvement on performance (in average) for a 4-way machine relative to the performance of the original code. Furthermore, the original version of code fails to scale adequately when we increase the way of the machine (less than 7% except for `toast`). In sharp contrast, the oracle version of code shows high performance increases when scaling the reference machine (specially for those benchmarks with high coverage, where we observe the potential for a 2.2X performance improvement).

2.3 Cost Effective Memory Disambiguation

We propose a software only mechanism that detects disjointed intervals at run-time and executes the corresponding disambiguated/non-disambiguated code. It consists on the following: for each modulo scheduleable loop, the compiler generates a copy of the loop code. In this copy, all dependence arcs between memory operations whose dynamic memory interval can be computed at run-time are deleted, unless static analysis indicates they are definitely dependent. Then, to ensure correctness, a test block is inserted before the two versions of the loop. The test block code computes the lower and upper bounds of each pair of dynamic memory intervals and compares them. It will conditionally branch to the original loop or to the disambiguated one depending on the result of the comparisons (see figure 4). Note that the test block is executed only once for each invocation of the loop, not for each iteration.

In most of the cases, the reduction on the schedule length of the disambiguated loop will be high enough to compensate the already low overhead involved in the calculation of the intervals. Otherwise, the penalty introduced by this block (if it turns out that the original loop is executed) is minimal, and has not a relevant impact on performance.

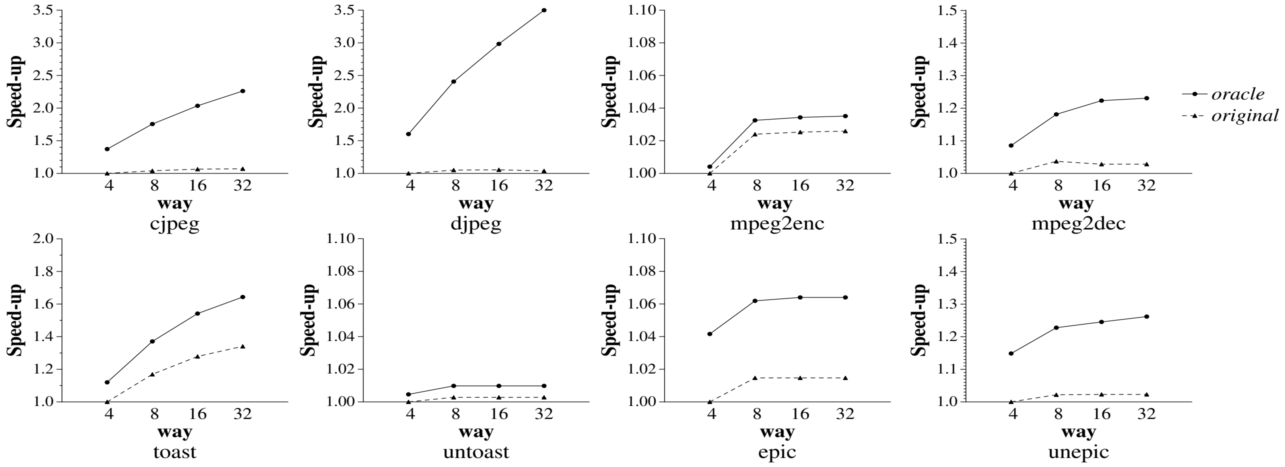


Figure 3: Potential speed-up

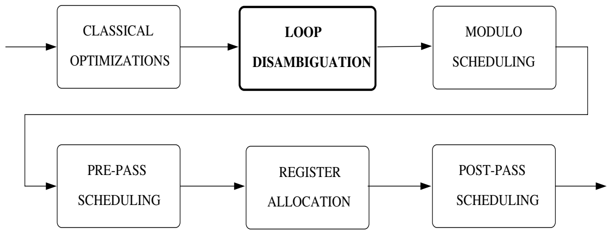


Figure 5: Phases of Elcor compilation

3. COMPILATION FRAMEWORK

In this section we discuss the algorithms built into Trimaran to support our memory disambiguation approach.

3.1 Evaluation Environment

Trimaran [11] is a compiler infrastructure for supporting state of the art research in compiling for Instruction Level Parallel (ILP) architectures. The system is currently oriented towards EPIC (Explicitly Parallel Instruction Computing) architectures. It is comprised by IMPACT, a compiler front-end for C, which performs machine independent optimizations, a compiler back-end (Elcor [1]) which performs machine-dependent optimizations, instruction scheduling and register allocation, and a cycle-level simulator of the parameterized architecture. To expose sufficient ILP it makes use of advanced techniques such as Superblock [15] or Hyperblock [13] formation. The architecture space is characterized by HPL-PD [10], a parameterized processor architecture.

Our compiler is representative enough of the current commercial compilers, and it does not use any interprocedural analysis. We have incorporated a new compilation module into the Elcor back-end to do loop disambiguation. Loop disambiguation is performed at the intermediate code level just before any scheduling or register allocation (see figure 5). It is executed in two phases. Firstly, the static disambiguator carries out some basic optimizations that were not included in the original compiler (see *3.1.2*). Secondly, the compiler applies the proposed run-time loop disambiguation method.

3.1.1 Terminology

To describe the implementation of the algorithm, we will use the following notation:
- *mop (memory operation)*: a load or store operation.
- N: number of iterations in one invocation of the loop.
- S_i *(stride)*: stride of the memory operation mop_i.
- AR_i *(address register)*: register containing the address referred by mop_i.
- DW_i *(data width)*: data size of mop_i in bytes.
- IW_i *(interval width)*: size of the memory region traveled by mop_i across all iterations.
- L_i *(lower bound)*: the lowest location referred by mop_i in one invocation of the loop.
- U_i *(upper bound)*: the next location to the highest one referred by mop_i in one invocation.
- $G_{\{i..j\}}$ *(group)*: set of memory operations from mop_i to mop_j. The concept of group is explained in section *3.1.2*.
- $GDW_{\{i..j\}}$ *(group data width)*: size of the memory region traveled by $G_{\{i..j\}}$ in one iteration.
- *define_op*(AR_i, $loop_l$): returns the operation that defines the register AR_i inside the loop $loop_l$.
- *is_dominator*(op_i, op_j): returns true if operation op_i is a dominator of operation op_j.
- *is_coincidence*(mop_i, mop_j): returns true if mop_i and mop_j refer the same location and have equal stride and data width.
- $edge_{ij}$: dependence arc from mop_i to mop_j.
- *del_memdep*($edge_{ij}$): deletes the dependence arc $edge_{ij}$.
- *is_modsched*($loop_l$): $loop_l$ can be modulo scheduled.

3.1.2 Basic Static Optimizations

We have added some static optimizations targeted to solve the inability of the original compiler to disambiguate some trivial situations, such as the loop (b) in figure 1. In that case, there are two coincident arrays, but different elements are accessed each iteration and thus cross-iteration dependences can be deleted. The algorithm to disambiguate them is shown in figure 6. If the compiler finds a pair of memory operations that refer the same locations with a stride greater or equal to the data size, it removes the cross-iteration dependence from the second operation to the first one.

It is quite common in multimedia applications to find a loop that has been partially unrolled by the programmer (see example in figure 7). At low level code, the different memory

foreach pair (mop_i, mop_j)
    if ($is\_dominator(mop_i, mop_j)$)
        if ($is\_coincidence(mop_i, mop_j)$)
            if ($S_i \geq DW_i$)
                $del\_memdep(edge_{ji})$
            endif
        endif
    endif
endfor

Figure 6: Algorithm for coincident memory operations optimization

addresses of the unrolled reference are computed by adding a different offset to a common base register. This situation can be detected at compile time and consider them as a single *group*. The stride of the whole group is just the same as that of each individual operation. Different locations are referred by each operation of the group, so we define the *group data width* as the size of the memory region from the lowest to the highest one. When a group of stores is found, the compiler can safely disambiguate them if the following two assertions are true: (a) the minimal difference between two offsets is greater or equal to the data size, and (b) the group data size is less than the stride of the group. In the example (figure 7), there are two coincident load and store groups. Our basic static optimizator would detect them and remove all dashed arcs in the dependence graph.

3.2 Algorithm Implementation

This section describes the main features of the algorithm implementation. It can only be applied to memory operations whose dynamic memory interval bounds can be computed at run-time previously to each execution of the loop. Thus, only dependences between strided or loop-invariant memory operations are selected for possible disambiguation. Dependences between operations which can be statically determined that refer the same location are also excluded, as they are definitely dependent. The compiler duplicates the loop and creates a new basic block just before both loops. This block contains the operations needed to test each of the selected dependences. Finally, the compiler removes the selected dependences in the dependence graph of the disambiguated version. The next subsection explains in more detail which operations must be inserted in the test block.

3.2.1 Building the Test Basicblock

Suppose an ambiguous memory dependence exists between two memory operations whose dynamic memory intervals are $[L_i, U_i)$ and $[L_j, U_j)$ (see figure 9.a). Then, to ensure they are disjointed intervals, we must test that:

$$(L_j \geq U_i) or (L_i \geq U_j)$$

where L_i, U_i, L_j and U_j can be computed in this way:

if ($S_i >= 0$)
    $L_i = AR_i$
    $U_i = AR_i + (N-1) * S_i + DW_i$
else
    $L_i = AR_i + (N-1) * S_i$
    $U_i = AR_i + DW_i$
endif

The compiler must insert explicit operations into the test block to perform the products and adds. Once this is done, the compiler inserts the compare and the conditional branch

```
for(i=0; i<N; i+=3) {
    ... = A[i];
    ... = A[i+1];
    ... = A[i+2];

    A[i] = ... ;
    A[i+1] = ... ;
    A[i+2] = ... ;
}
```

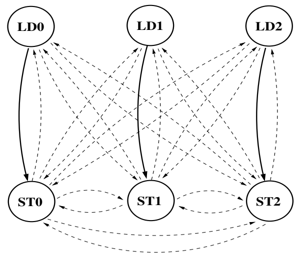


Figure 7: Example of coincident group

foreach $edge_{ij}$ *to disambiguate*
    if ($is\_dominator(define\_op(AR_i, loop_l), mop_i)$)
        $insert\_previous\_ops(mop_i)$
    endif
    if ($is\_dominator(define\_op(AR_j, loop_l), mop_j)$)
        $insert\_previous\_ops(mop_j)$
    endif
    $insert\_interval\_computation\_ops(mop_i)$
    $insert\_interval\_computation\_ops(mop_j)$
    $insert\_compare\_intervals\_ops(mop_i, mop_j)$
endfor
$insert\_branch\_op()$

Figure 8: Algorithm for building the test block

operations. The steps to create the test block that selects either of the two loops are summarized in Figure 8.

The *insert_previous_ops* function takes into account the case in which the address register is defined inside the loop body before being used by the memory operation, that is, when the value of the register AR before entering the loop is not the value it will have when the memory operation will be executed in the first iteration. In that case, the compiler must also insert an equivalent copy of the define operation before the bounds computation in order to get the right value of AR. In this copy, the register is renamed to avoid modifying the real value. Note that *insert_previous_ops* is a recursive function, as now it must ensure data dependences are maintained for each operand of the define operation. The function *insert_interval_computation_ops* creates the low level products and additions to compute L and U, and *insert_compare_intervals_ops* inserts the operations to compare these limits. Finally, *insert_branch_op* inserts the conditional branch.

3.2.2 Optimizations to reduce the length of the test block

The length of the test block should be controlled not only because of performance, but also because it increases the register pressure. For our study, we have used a simple heuristic that limits the maximum number of dependences to be tested. More effective heuristics that take into account the schedule length of the loop should be analyzed.

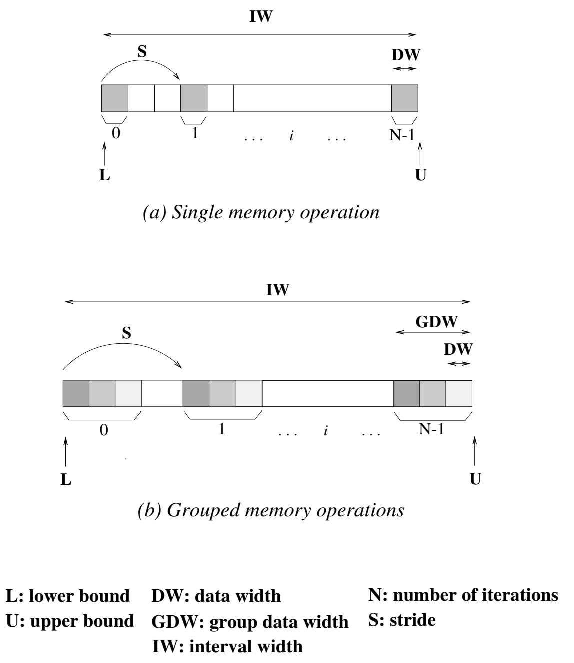


Figure 9: Dynamic Memory Intervals bounds representation

According to the previous section, for each pair of intervals to be compared, we would need at most two products, six adds and two compare operations. Such a quantity of operations could become prohibitive as the number of dependences to be tested increases. However, they can be reduced if we take into account some trivial considerations, such as: operations with the same stride share a single product, the multiply operation can be replaced by a shift operation if the stride is a power of two, or if a memory reference must be compared with more than other one, the interval bounds are computed just once, so that only the compare operations need to be added.

One important optimization to reduce the test block is to make groups of memory operations. As said in *3.1.2*, partially unrolled references are frequent in multimedia codes. The compiler can group these references and consider them as a single widener memory operation (figure 9.b). The group data width can be computed as the difference between the offsets of the last and the first elements plus the data width. The algorithm to compute the interval bounds remains the same except that now the group data width is used instead of the single data width:

$$
\begin{array}{l}
\text{if } (S_{\{i..j\}} >= 0) \\
\quad L_{\{i..j\}} = AR_i \\
\quad U_{\{i..j\}} = AR_i + (N-1) * S_{\{i..j\}} + GDW_{\{i..j\}} \\
\text{else} \\
\quad L_{\{i..j\}} = AR_i + (N-1) * S_{\{i..j\}} \\
\quad U_{\{i..j\}} = AR_i + GDW_{\{i..j\}} \\
\text{endif}
\end{array}
$$

This optimization achieves high reductions on the length of the test block. For example, in the color conversion function of `cjpeg`, for each pixel of a row, the three color components (R, G and B) are loaded each iteration (see code in figure 10.a). These color components are one byte size and are interleaved stored in memory:

$$R_0 G_0 B_0 R_1 G_1 B_1 R_2 G_2 B_2 R_3 \ldots$$

Without grouping, we would have three intervals with stride three and data width one byte, and each interval should be compared with the rest of memory operations. However, if the compiler realizes they form a group, it can consider one interval with stride three and group data size three (see figure 10.b), and save an important amount of arithmetic and compare operations. The effect is still greater for loops with large groups, such as the DCT computation, where eight memory operations are grouped to form a single group.

The code of the test block inserted by the compiler to disambiguate the three loads and the three stores of the color conversion loop body is given in figure 10.c. Let us assume that registers $r3$, $r4$, $r5$ and $r7$ are the address registers concerning to $outp0$, $outp1$, $outp2$ and inp respectively, and the control register LC ($loop\ counter$) contains the number of iterations. Then, operations from 3 to 12 are inserted to compute the interval bounds ($r3$, $r4$, $r5$ and $r7$ are the lower bounds and $r13$, $r14$, $r15$ and $r17$ the upper ones).

Once the interval bounds are computed, the compiler introduces the operations needed to check if they overlap (operations 1 and from 13 to 24). To support predicated execution, the HPL-PD architecture provides 1-bit predicate register files and a rich set of compare-to-predicate operations which set predicate registers [10]. We make use of these capabilities to generate the code of the test block. In the example, predicate registers are denoted as "pn". The OR-compare operations (e.x., $p2 \mid = (r7 < r15)\ if\ p8$) write a 1 into the destination register ($p2$) only if both the predicate input ($p8$) and the result of the comparison are true. Otherwise, they leave the destination unchanged. These kind of operations are useful in the efficient evaluation of boolean reductions, as any OR reduction can be evaluated in a time equal to the latency of a single compare operation provided there is enough parallelism in the machine.

The conditional branch is performed in two steps. Firstly, the prepare-to-branch (PBRR) operation loads the target address into a branch-target register ("btr$n$"). Secondly, the branch-conditional (BRCT) operation branches to the address contained in the btr$n$ operand if the branch condition (available in the specified predicate) is true. In the example, the branch-target register $btr4$ holds the address of the non-disambiguated loop version, and operation 25 branches to it if the result of the comparisons is true, that is, if any of the dynamic memory intervals pairs overlap.

3.2.3 *Loop Memory Disambiguation Based on Profiling*

To further reduce the number of failed tests, we could use profiling information to select which dependences are the candidates to be disambiguated. The profiler must check if the dynamic memory intervals of each pair of memory references overlap or do not overlap and produce a list of pairs of disjointed memory operations.

For each loop which can be modulo scheduled, the loop disambiguation tool takes as input the list produced by the profiler (instead of building by itself a list of all dependences between non coincident operations) and proceeds as usual (see algorithm in figure 11). Pairs in which any of the two references are neither strided nor loop invariant are discarded, as their dynamic memory interval cannot be computed. If one or more pairs of references remain in the list, the compiler duplicates the loop and inserts the test block. Finally, dependences arcs in the list are removed in the disambiguated loop version.

(a) C code fragment:

```
inp = *input_buf++;
outp0 = output_buf[0][output_row];
outp1 = output_buf[1][output_row];
outp2 = output_buf[2][output_row++];

for (col = 0; c < ncols; col++, inp += 3)
{
   outp0[col] = (K1*inp[0] + K2*inp[1] + K3*inp[2] + ... )
   outp1[col] = (K4*inp[0] + K5*inp[1] + K6*inp[2] + ... )
   outp2[col] = (K7*inp[0] + K8*inp[1] + K9*inp[2] + ... )
}
```

(c) Test Block code:

```
CLEAR p2
btr4 = PBRR BB_27
LC = LC - 1
r19 = LC * 3
r10 = LC + r3
r11 = LC + r4
r12 = LC + r5
r16 = r19 + r7
r13 = r10 + 1
r14 = r11 + 1
r15 = r12 + 1
r17 = r16 + 3
p3 = (r3<r14)
p4 = (r3<r15)
p5 = (r4<r15)
p6 = (r3<r17)
p7 = (r4<r17)
p8 = (r5<r17)
p2 |= (r4<r13) if p3
p2 |= (r5<r13) if p4
p2 |= (r5<r14) if p5
p2 |= (r7<r13) if p6
p2 |= (r7<r14) if p7
p2 |= (r7<r15) if p8
BRCT btr4 if p2
```

(b) Dynamic Memory Intervals:

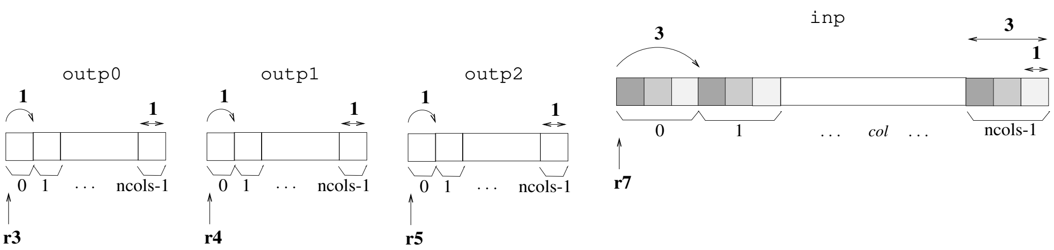


Figure 10: Example of test block code generated by the compiler

```
foreach loop_l
   if (!is_modsched(loop_l))
      continue
   endif
   foreach edge_ij in profiler_list(loop_l)
      if (!is_strided(mop_i) and !is_invariant(mop_i))
         remove_from_list(edge_ij)
      endif
      if (!is_strided(mop_j) and !is_invariant(mop_j))
         remove_from_list(edge_ij)
      endif
   endfor
   if (1 < list_size < MAX_SIZE)
      create_loop_copy(loop_l)
      create_loop_test_block(loop_l)
      foreach edge_ij in list
         del_memdep(edge_ij) in loop_copy
      endfor
   endif
endfor
```

Figure 11: Algorithm for loop disambiguation based on profiling

4. EVALUATION

In this section we provide quantitative data to show the effectiveness of our proposal. Additionally to performance results, we will study the impact of our technique over the code size and the compilation time.

The design of most of the current generation of DSPs is based on VLIW architectures [24, 5, 26]. We have simulated a VLIW architecture with predicated execution. Hyperblock formation is performed by the Impact front-end. During hyperblock formation, if-conversion [19] is used to form larger blocks of operations and thus provide a greater opportunity for code motion to increase ILP. For a I-way machine, up to I operations can be embedded into one VLIW-instruction, but the number of operations of each type is limited up to I integer operations, $I/2$ memory operations and one or none branch operation. Different inputs have been used for training and evaluation when using profiling.

4.1 Performance

Figure 12 shows performance results for the selected benchmarks. The original compiler plus our static optimizations (second bar in the graphs) has been chosen as the baseline. Each bar represents speed-up versus the 4-way baseline. Next two bars are our approach without and with profiling assistance. The last one is the ideal case in which all independent references are disambiguated for all modulo scheduleable loops. For clearness, different yaxis scales have been drawn for each benchmark.

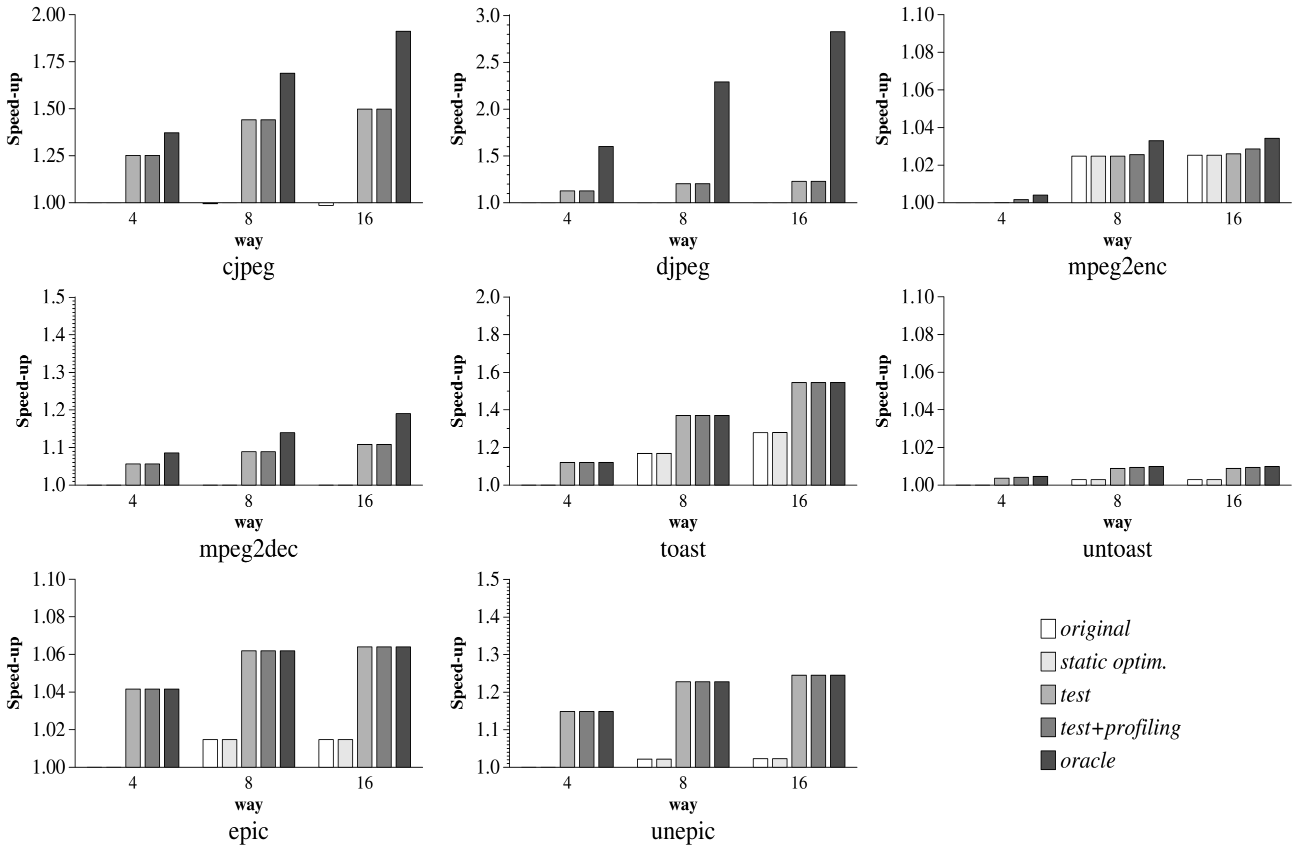


Figure 12: Speedups

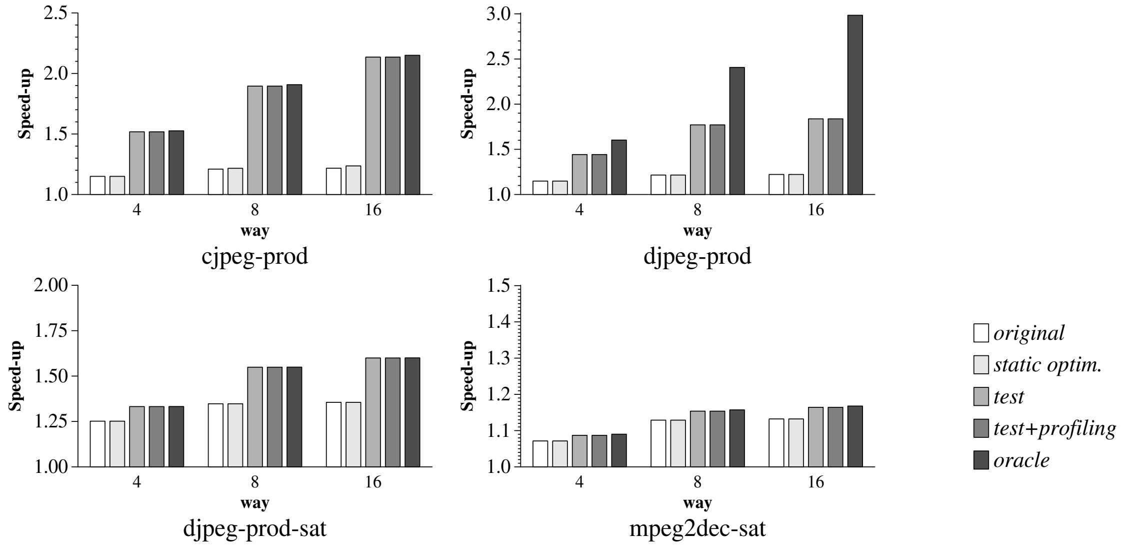


Figure 13: Speedups of optimized versions

As can be observed, in general, our technique is quite close to the oracle. For a 4-way machine, speed-ups up to 1.25X (`cjpeg`) are achieved. Moreover, in sharp contrast with the baseline configuration, our technique scales adequately when we increase the way of the machine (up to a 23% of performance increase when using a 8-way machine instead of a 4-way machine).

The fact that the profiling-version of the technique obtains nearly the same results than the non profiled one demonstrates our assumption that a high percentage of the loop candidates disambiguate (which supports the assumption that media loops are characterized for high amounts of Data Level Parallelism). The only benchmark that shows a difference (while minimally) is `untoast`. This benchmark has a significant loop with a direct recurrence, thus, making the block test inserted in the non profiled version to fail in every instance, and paying as a result a penalty of seven cycles per each execution of the loop. However, test blocks represent a negligible overhead compared with the rest of the code. In general, we have observed that the test blocks do not represent more than a 0.1% of the overall dynamic instruction count of the loops. The only exception is in `mpeg2enc` where we have found one test block that represents roughly a 10%. Even for this last test block, results pay off as we are saving more execution cycles from the optimization of the target loop when doing modulo scheduling.

A very important observation from results of figure 12 comes from the effective speed-ups achieved by our technique compared with the oracle predictions for some of the benchmarks. While our technique provides good improvements (1.25X in `cjpeg`, 1.13X in `djpeg` and 1.06X in `mpeg2dec` for the 4-way architecture), we are still significantly far from the ideal case. The reason that explains such a phenomena is the use of non-streaming (sparse) data structures to perform computations via memory tables. This effect is studied in next section.

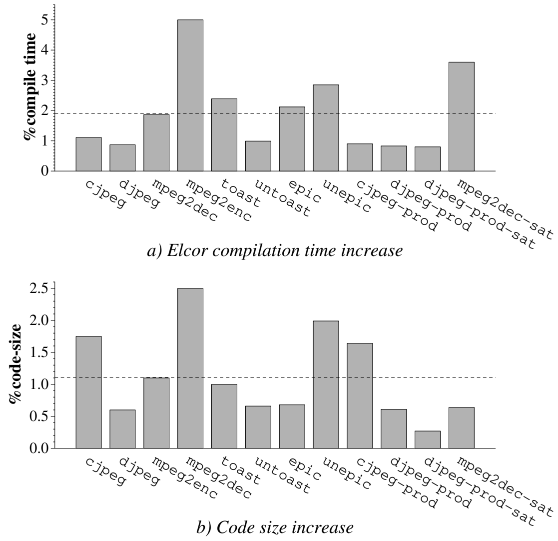
4.1.1 *Effect of DSP Oriented Scalar Optimizations*

Most of the algorithms used in the current standards have a streaming nature. Nevertheless, due to the intrinsic significance of most multimedia algorithms, there has been a great effort focused on reducing the overall number of required operations. Unfortunately, this effort has been oriented towards low-end DSP architectures, thus focusing on reducing the number of operations or, more specifically, the number of high latency operations such as multiplications, rather than on making explicit the potential ILP of the algorithms[23].

For example, in `cjpeg`, the color conversion function uses memory tables to replace multiplications. The table references cannot be disambiguated, as they do not have strided patterns. This loop stands for a 31.5% of the overall number of dynamic instructions in the program and it is the cause of the gap between our proposal and the oracle. A similar case occurs with saturation (clipping a result to a maximum/minimum value if it exceeds a given range), which is also implemented using memory tables in `djpeg` and `mpeg2dec`.

We are interested in evaluating the performance of our technique when we revert to the original ways of performing the computation. Thus, we have analyzed the following explicit parallel versions:

- `cjpeg-prod`: use explicit products to perform color conversion instead of the tables.



a) Elcor compilation time increase

b) Code size increase

Figure 14: Cost of the loop disambiguation method

- `djpeg-prod`: use explicit products to perform the inverse color conversion.
- `djpeg-prod-sat`: same as `djpeg-prod`, but saturation is also implemented without tables.
- `mpeg2dec-sat`: saturation tables are replaced by the explicit if condition.

Figure 13 shows the same speed-up measurements for the new versions of code. The fact that `djpeg-prod` outperforms `djpeg-prod-sat` is due to the Hyperblock formation mechanism used by Impact, which relies on profiling information, so that non-executed paths are not included in the hyperblock. This makes Elcor fails in doing modulo scheduling to one of the most time-consuming loops of `djpeg-prod-sat`. Two conclusions can be inferred from the results. First, we can observe that the new versions of code achieve significantly higher overall speed-ups than the original versions of code, as the ILP is now explicit for the compiler. Second, the performance results leveraged by our technique are now significantly closer to the ideal.

4.2 Code Size and Compilation Time

The use of the proposed method involves an increase in code size and compilation time. However, the optimization is only applied to a minimal part of the static code, so the overall impact is negligible. The first graphic in figure 14 shows the percentage of additional time required by Elcor to perform machine dependent compilation. Both, static and dynamic optimization times are included. As it can be seen, the increase is often less than 2%. It refers only to the Elcor module compilation, that is, the time required by the front-end Impact is not included (as it does not vary), and thus, the effect on the overall compilation time is even lower. As far as code size is concerned (figure 14.b), the increase in the number of static operations ranges from 0.27% to 2.51%, with an average of 1.19%. Thus, we could affirm that the improvement on performance is in practice obtained for free.

5. CONCLUSIONS

Memory disambiguation of multimedia applications is compromised by the fact that they are often written in languages that support pointer referencing, such as C or C++. In this paper we have presented a simple but efficient memory disambiguation technique specifically targeted at multimedia loops. Taking into account the disjointed behavior of common media memory streams, our algorithm is able to evaluate at run time whether or not the full loop is disambiguated and execute the corresponding loop version. By calculating at run-time the dynamic memory intervals of every memory reference in a very efficient way, we avoid having to perform comparisons inside every loop iteration.

The main features of our technique are the following:

- Significant performance gains for a 4-way VLIW configuration for most of our benchmarks.

- Performance scalability of wider-issue machines (up to 16-way) in sharp contrast with our baseline.

- Near-zero cost for all those loops without potential for disambiguation.

- Negligible effects over compilation time and very low code size increases.

- Simple and fast implementation.

Future work includes the evaluation of the algorithm on a broader range of multimedia applications, together with the study of the use of better heuristics to decide when it is worth or not to do loop disambiguation. We also plan to extend our technique to calculate the dynamic memory interval of operations related to tables.

6. REFERENCES

[1] S. Aditya, V. Kathail, and B. R. Rau. Elcor's machine description system: Version 3.0. Technical Report HPL-98-128, Information Technology Center, 1998.

[2] D. Bernstein, D. Cohen, and D. E. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 105–111, November 1994.

[3] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[4] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[5] A. Devices. Introducing tigersharc. Whitepaper. Available at http://www.analog.com/new/ads/html/SHARC2.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[7] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. *ACM SIGPLAN Notices*, 29(11):183–193, 1994.

[8] G. Golf, K. Kennedy, and C. Tseng. Practical dependence testing. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[9] A. Huang, G. Slavenburg, and J. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 200–210, April 1994.

[10] V. Kathail, M. Schlansker, and B. R. Rau. Hpl-pd architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett–Packard Laboratories, 2000.

[11] H. P. Lab., R.-I. Group, and I. Group. Trimaran user manual. Available at http://www.trimaran.org/docs.html, 1998.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *MICRO-30*, 1997.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.

[14] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[15] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, , and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing, Kluwer Academic Publishers*, pages 229–248, 1993.

[16] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *ACM Symposium on Principles Practice of Parallel Programming*, pages 84–95, 1999.

[17] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[18] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. *ACM SIGPLAN Notices*, 33(5):60–71, 1998.

[19] J. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett–Packard Laboratories, 1991.

[20] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.

[21] B. R. Rau. Iterative modulo scheduling. Technical Report HPL-94-115, Hewlett–Packard Laboratories, 1995.

[22] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, 1994.

[23] E. Salamí, J. Corbal, R. Espasa, and M. Valero. An evaluation of different dlp alternatives for the embedded media domain. *1st Workshop on Media Processors and DSPs, in conjunction with MICRO32*, 1999.

[24] P. Semiconductors. Trimedia tm-1300. Available at http://www-us3.semiconductors.com/trimedia/, 1999.

[25] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[26] TI. TMS320C62XX family. Technical Report .Available at http://www.ti.com/sc/docs/products/dsp/tms320c6201.html, Texas Instruments, 1999.

[27] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–, 1995.