Towards Optimal Instruction Vectorization

Steven Osman and Ryan Williams

April 28, 2003

1 Introduction

Some modern processors allow the same arithmetic or logical operation (for example, an addition) to be performed on multiple values using one instruction and wide registers [5, 8]. For example, a 128-bit register can be configured to store four 32-bit values, all of which are added or multiplied simultaneously. Not only can this lead to a more efficient program (one instruction can execute more quickly than four), but by packing multiple values into a single register it also allows better register utilization. The Sony Playstation 2 game console, our focus in this project, is configured with a modified Toshiba 5900 MIPS processor and is capable of numerous SIMD (Single Instruction, Multiple Data) instructions with 128-bit wide registers (the Emotion Engine). Under this instruction set, it is possible to parallelize additions, subtractions, logical operations, and even loads and stores (if the individual loads/stores are guaranteed to access consecutive memory locations).

A natural question we have asked is: given code that does not use these SIMD instructions (or uses them rarely), in what cases may we replace a few non-SIMD instructions with their SIMD counterpart, to optimize performance? That is, at what program points is it beneficial to *vectorize* a set of instructions into one?

Intuitively, it would appear that such optimizations would be applicable quite often in multimedia applications, where matrix algebra operations such as linear transformations are abound. Operations on arrays seem as if they would be quite susceptible to this idea of vectorization. A simple instance is the initialization of an array: it could be done faster, if we initialized several of the entries at once using SIMD instructions. A linear transformation, where a series of data-independent dot products are performed on the rows of a matrix, is similarly susceptible to vectorization.

2 Initial Problems

2.1 Design

An immediate problem one encounters with this kind of optimization is the overhead incurred in setting up the wide register– if four instructions are vectorizable, it may still be that while

performing the corresponding SIMD instruction is faster, there is a non-negligible cost in initializing the wide register and loading the resulting values back into the short registers. To be perfectly clear, we give an example.

```
a1 ← b1 + c1
a2 ← b2 + c2
a3 ← b3 + c3
a4 ← b4 + c4
```

Suppose we choose to vectorize these four instructions into one. Then the resulting code, not assuming any particular program context, would have the form:

```
b ← vectorize(b1,b2,b3,b4)
c ← vectorize(c1,c2,c3,c4)
a ← vec-add(b,c)
a1 ← vec-load(a,1)
a2 ← vec-load(a,2)
a3 ← vec-load(a,3)
a4 ← vec-load(a,4)
```

The difficulty is that these `vectorize` and `vec-load` operations typically require more time than that saved due to the vector addition. To avoid problems of this nature, we restricted ourselves to optimizations that can be performed inside of loops. In particular, we looked for optimizations where the arithmetic/logical vector operation can be performed inside of the loop, while the `vectorize` and `vec-load` operations can be performed outside of the loop body (immediately before and immediately after the loop, respectively).

2.2 Implementation

A number of problems were encountered while developing a prototype for this algorithm. First, we weren't as familiar with GCC as we were with SUIF. After exploring a SUIF implementation for the IA32 architecture, we found the IA32 multimedia instructions a little lacking when compared with the Emotion Engine's instructions. A more severe problem we encountered was that our algorithm, as it stands, does not allow for a dependency on the loop counter induction variable. This, unfortunately, prevents us from vectorizing array accesses which is what the vector instructions were optimized for. We believe, however, that the constraints imposed on the algorithm can be relaxed to accommodate this situation.

3 Previous Work

As the problem of vectorizing code optimally is quite hard, the prior work in vectorization always works in limited cases, and is generally quite conservative in its scope. We highlight the results of three references here; other references include [7],[2]. DeVries' work [2, 3] on developing a vectorizing SUIF compiler gives some interesting techniques, but he only applies them to instructions with array references, and he applies them all over the code. However, this optimization is susceptible to the problems we mentioned previously.

A 15-740 course project from a couple of years ago [4] explores opportunities for vectorization. Along the way they define a graph similar to ours, but the optimization they perform is *extremely* time inefficient (see next section).

Lee and Stoodley [6] focus on the task of vectorizing within loops; as we have stated, we believe is the best direction to take. More precisely, they show how parallelizing outer loops using vectorization can lead to a speed-up. In this situation, their optimization is effective but a bit restrictive– it only applies to outer loops and no other control structures.

In constrast to the above, our work focuses solely on optimizing natural loops without breaks in them. Within a loop, we restrict our attention to a set of variables we will call the candidate set. We argue our restriction is not major (i.e. it still applies to most cases of a loop); however, imposing it makes the problem of optimal vectorizing immensely more tractable.

4 Overview of Optimizations

We took two directions in designing loop optimizations. The first was to try to identify cases where we could vectorize collections of definition points within the loop, according to a set of general conditions on these variables. The algorithm we have devised for doing this is quite natural and involves an interpretation the pattern of uses and definitions in the loop as a directed graph. In retrospect, our approach is quite similar to that of [4], with the notable exception that our optimization always results in graphs that can be analyzed efficiently.

In the absence of such optimization conditions, the second idea was to unroll a loop for a few steps in the hopes of creating possibilities for vectorization via the algorithm. We did not make much progress on this idea, but will discuss it later nevertheless. The two approaches invite a natural comparison with *global redundancy elimination* and *partial redundancy elimination*, where in the latter we modify code to make the former possible.

4.1 Vectorization

We have assumed throughout this project that there is a high overhead incurred from "loading" and "unloading" wide registers with the contents of smaller registers, i.e. the overhead of vectorization is high. Hence, when we decide to vectorize a group of definition points

(variables) within a loop, we will commit to vectorizing *all* operations that involve these variables. Otherwise, if we had a short register and a wide register with one of its components corresponding to the same variable, the cost of updating the two registers may be high.

Thus we will talk about *vectorizing groups of variables* in a loop. In other words, we replace all the instructions defining and using these variables with analogous parallel instructions.

The major impediment to vectorization of a set of operations concerns the variable usage within the loop. For example, if i is a basic induction variable used to detect when to end the loop, then if we vectorize the add operations for updating i, each time the comparison with i is performed we must extract the current value of i from the wide register in order to perform the comparison, which can be costly.

As a whole, data dependency can make a piece of code very unfriendly to vectorization. If we had code of the form: $c = a + b$, $a = d + e$, $b = f + g$, etc., in order to vectorize these variables, we'd need to have at least three different vectors, so that c, a, and b are in the same components of the vectors. Similarly, we'd need d, e, f, and g to be in the same components as a and b; this implies we'd need at least seven different vectors in order to vectorize this code. So clearly, there are cases where vectorization is a very bad idea, and it is not detected easily via previous approaches. Our algorithm can be tuned to disregard what we will call large *use graphs*, which correspond exactly to the above scenario.

4.2 Main Algorithm

We now describe our main algorithm for parallelizing instructions, which is applicable in many situations. In particular, the optimization applies to any set of variables S in a loop where:

(a) All of the variables, if defined in the loop, are defined with respect to some operation $\oplus$ with a right identity element e (i.e. $x \oplus e = x$ for all x)[1],

(b) All of them have at most one definition point in the loop, and

(c) All of them have at most one *definition-point usage* somewhere in the loop. More precisely, for each variable, either its usage is in a definition for some variable in S, or it appears is in the definition of a variable that's not used within the loop. (A definition does not count as a usage.)

We will call such a set S a *candidate set*. What the optimization will do is vectorize the variables in S– piece them together into vectors of whatever constant size you choose– so that the overhead of loading/unloading can be placed outside of the loop, and all of the operations done on the variables of S inside of the loop are parallel ones. The algorithm takes time quadratic in the number of registers, and is described next.

[1]Note that all arithmetic operations and logical operations (including subtraction) have a right identity. We need a right identity because there is no left identity for subtraction, i.e. there is no e where $e - x = x$ for all x.

4.3 Algorithm

We give an English description of the procedure.

For each natural loop L, do the following.

4.3.1 Building a set of candidates

Initially, the candidate set S is a list of all registers. For each register $r \in S$, count the number of uses and definition points for r in the loop L. If the number of uses or the number of defs for a register is ever greater than 1, remove r from S.

Then repeat until no r are removed (or iterate over the number of registers):

For each $r \in S$ with a definition point p in L, if there is an operand o in p such that $o \notin S$, then remove r from S.

After this step, every $r \in S$ has at most one definition point in L and is used at most once in L. Furthermore, the operands used in its definition point (if it has one) are registers which are also in S.

4.3.2 The use graph

To ease the conceptual idea of vectorization, we define a directed graph we call a *use graph*, which represents the pattern of uses and definitions in the loop L. The vertex set V will be the set of all constants and registers used by definition points in S. If R is the number of registers, we make R distinct 'copies' of the constants and registers used in the vertex set. (Note $S \subseteq V$.) For each operation $\oplus$ used in L, we associate a color $c(\oplus)$.

Then for $v_1, v_2 \in V$, we put a directed edge of color $c(\oplus)$ from v_1 to v_2 if v_1 has a definition point via operation $\oplus$ in the loop, and v_1 uses v_2 in that definition. If v_2 corresponds to a constant or register, we put an edge from v_1 to a fresh 'copy' of v_2. Observe that every v_i has either no outgoing edges, or two outgoing edges (either it is not defined, or it is). Also, observe that v_i could have an edge to itself (a definition point in which it uses itself), in which case it has no edges to any other nodes in the graph.

Looking back at previous work, our definition is similar to that for an IODL dependency graph [4], but their graphs in general are much more complicated, and they do not associate colors with edges, which is a restriction that makes graph comparison easier. The candidate set we have chosen for performing this optimization will permit us to vectorize these operations *far* more efficiently (the optimization in [4] takes $O(n!)$ time in general, where n is the number of instructions).

4.3.3 Identify strongly connected components

Consider each definition point p corresponding to a register $r \in S$ in the order they are presented in L. Find the strongly connected component C_r of r in the graph– i.e. all of the

registers and constants in S that have a data dependency with r. Remove from consideration all registers and constants with this dependency. Do this until each $v \in S$ appears in one of the strongly connected components. This can be done in time linear in the number of registers.

4.3.4 Grouping together components for vectorization

We will vectorize by partitioning together strongly connected components into groups. The size of the groups is equal to the number of components in the vectors (in our case, four). In each group, we will pick unique registers from each strongly connected component of that group– these registers will be combined into a vector. However, not all possible groupings can be done: we want each register in the original program to be included in exactly one vector in the new vectorized code (otherwise, we could have problems figuring out which "version" of the register to use). If the strongly connected components do not "match up" in the proper way, we will not be able to achieve this.

To create these groups from the connected components, we take a greedy approach. We will say $C \leq C'$ if the connected component C is a *colored subgraph*[2] of C'. Start with the largest connected component C. Find the three largest other connected components C_1, C_2, C_3 such that $C_1 \leq C_2 \leq C_3 \leq C_4$, and place them in the group with C. Remove all four components from consideration and pick the next largest connected component. We repeat this process until there are no groups of four, and then perform it on groups of three, then two. After this, we may conclude that none of the remaining graphs are subgraphs of another, in which case we put all of them in separate groups.

We remark that given the out-degree of any vertex is either two or zero, and the in-degree is at most one, determining whether or not one component is a colored subgraph of another can be done in linear time.

4.3.5 Vectorizing using the groups

Finally, given these groups, we will vectorize using the subgraph property found in the previous stage. Let's first consider the case where the group is of size 2. Recall that we put C and C' in the same group because one of them is a subgraph of another; say C' is a subgraph of C.

One technical detail needs to be ironed out, which will be clearer when examples are seen. Our vectors have quantities which change in the loop, and those that do not change. When an operation happens to update the changing quantities, we need to do some kind of "no-op" on the constant values, to preserve them. The vertices with out-degree zero directly correspond to these quantities that are constant throughout the loop.

[2]Informally, by colored subgraph we mean that not only do the edges and vertices "match up", but the colors of edges do as well.

Define V' to be the set of vertices $v \in C'$ where v has out-degree zero and $h(v) \in C$ has out-degree greater than zero. We define a map h from the vertices in C to those in the subgraph C' as follows:

1. If v' is the vertex in C' corresponding to v in C (as a subgraph of C'), then $h(v) := v' \in C'$.

2. If no vertex in C' corresponds to $v \in C$, but the parent p of v in C is such that $h(p)$ has out-degree 0:

   If v is the right child, then define $h(v)$ to be the identity of the operation.

   If v is the left child, then $h(v) := h(p)$.

3. Otherwise, $h(v)$ is defined as the identity of the operation.

Now, prior to the loop, we insert code that initializes a vector:

$$V_i := [r_i, h(r_i)], \text{ for each } r_i \in C.$$

Next, within the loop body, for each $r_i \in C$ (consider them in order of definition point), we delete the instructions defining r_i and $h(r_i)$. By the fact that C' is a subgraph of C, if $r_i = op(r_j, r_k)$ then $h(r_i) = op(h(r_j), h(r_k))$ (note they must be defined according to the same operation). Then we insert at the earliest place (either the definition point of r_i or the definition point of $h(r_i)$) the vectorized instruction:

$$V_i := parallel_op(V_j, V_k).$$

Let's briefly look back at step 2 of h's definition and explain its meaning. There, we simply ensure that when r_i residing in vector V_i gets re-defined via the operands r_j and r_k, the second component of V_i (the constant quantity $h(r_i)$) is still $h(r_i)$ after the operation. That is, $op(h(r_j), h(r_k)) = op(1, h(r_i)) = h(r_i)$, where 1 is the identity of the operation.

Finally, after the loop, we insert code that extracts the registers from the vectors we have constructed:

$$r_i := V_i[0], \text{ for each register } r_i \in V_i, \text{ and}$$
$$h(r_i) := V_i[1], \text{ for each } h(r_i) \text{ that is a register.}$$

For groups of larger size, the generalization of this above vectorization is fairly straightforward: we initialize a vector $[r_i, h_1(r_i), h_2(r_i), \ldots]$, where h_j maps from the larger component C to a subgraph C_j in the group. In the case of loop-invariant values, we ensure that one of the h_j maps to the correct $h(p)$, and the others map to the identity of the operation, so that when an operation is carried out, $h(p)$ is returned.

5 Optimization Examples

Here's a simple example with a self-loop. Consider the piece of code in a loop body.

```
sum=sum+a
```

```
a=b+c
```

```
f=f+g,
```

where b, c, and g have no definition points in the loop.

Assume that the registers with at most one definition point and at most one use (and its operands in its definition points have at most definition point and at most one use) have been identified, and they are the above variables.

The use graph is then:

```
*sum -> a -> b
        |
        v
        c

*f -> g
```

(* represents a self-loop).

Assuming b is the left child and c is the right child of a,

c is initialized with 0 and b is initialized with g, resulting in:

```
[sum,f]=[sum,f]+[a,g]
```

```
[a,g]=[b,g]+[c,0].
```

5.1 An interesting toy example

Suppose the loop is:

```
..(start of loop)..
a=b+c
b=d+e
c=a*f
g=1+h
h=g*2
..(end of loop)..
```

Our algorithm transforms this code to (in slightly abusive pseudocode):

```
A = [a,g]
B = [b,1]
C = [c,h]
D = [d,1]
E = [e,0]
F = [f,2]
..(start of loop)..
```

```
A = B + C
B = D + E
C = A * F
..(end of loop)..
a = A[0]
g = A[1]
```
etc.

The use graph is:

```
a -> b -> d
|       \
c -> f    e

g -> 1
|
h -> 2
```

(a and c have edges to each other, and so do g and h)

The algorithm computes these two graphs, finds that the second graph is a subgraph of the first, and vectorizes accordingly: i.e. g gets matched up with a, h is matched up with c, etc. For those vertices in one graph, where there is a definition (outgoing edges) but the corresponding vertex in the other graph doesn't have a defintion (no outgoing edges; in this case, b and 1), we have to put the 1 in one of the vectors for d and e, to "save" its value.

6 Implementation

We have been working with a version of GCC 3.0.3 that was ported to the PlayStation 2. There certainly was quite a bit of a learning curve involved in using GCC, however, the loop functionality is quite well documented which was a tremendous help.

We modified the loop functionality (in loop.c) at the two locations where unrolling would have occurred. If the compiler finds that it can vectorize a loop then it inhibits any further unrolling of the loop. This conservative step may not have been necessary but we did not have enough time to check.

We also modified the unrolling functionality (in unroll.c) to allow us to pre-specify the exact amount by which we want the loop unrolled. Refer to the Loop Unrolling section for more details.

6.1 Graph Matching

In order to define the best set of vectorizable instructions, it is important to find the largest set of matching graphs. This is complicated by the associativity of a lot of operations,

allowing us to try to compare different graphs in different configurations. We have considered a number of options to detect whether one graph is a subgraph of another, including [9] and the string matching method described in the implementation section. Even though these methods are efficient ways of detecting whether there is the *potential* for vectorization, the solution may still fail after a more fine-grain search (particularly with the string matching method). Also, once the match has been definitively decided, another difficult task is to rearrange one or both of the graphs so that they can overlay each other. Once again, this is because the associativity of many instructions give us the freedom to choose which order to lay out the operands in, however, this also requires searching the larger solution space for the ideal configuration.

We implemented our graph matching in three steps. Unfortunately, this implementation probably is very conservative in some cases, not identifying potential for vectorization where there may have been some.

First, nodes in the graphs were labeled with a textual representation of the graph. A rough approximation[3] of this is:

```
set(reg, add(set(reg, mult(reg, const), const) )
```

For nodes that have multiple children and commutative operators, (for example, an addition requires two operands), the children were sorted in order of their textual representation. This was to help ensure that the commutative operands were placed in an order that would maximize string matching across different graphs.

Second, we colored the graphs so that all disjoint nodes have different colors (i.e. each cluster of nodes has one color).

Finally, in order to pair two graphs together as vectorization candidates, we ensured that, for every node of the first color, there existed a unique node of the second color with the same string, and vice-versa.

The string comparison method could be used to vectorize array access as well with the following changes:

1. The string would actually contain the memory address (generally the number of the register that it is stored in).

2. The address register is not allowed to change after its first use.

3. The string matcher is adjusted to match strings such that a reference to memory X can only be matched by a reference to memory X+4 (for 32 bit values). Similarly, X+4 can be matched by X+8, and X+8 by X+12 for four-way vectorization.

For example:

```
set(reg, mem(regXXX))
```

would vectorize with

```
set(reg, mem(add(regXXX, const)))
```

[3]The actual representation is not human-readable

IFF const = 4

We have been able to create, color, and codify the dependency graphs (i.e. use graphs) for loops into sorted strings. Any two graphs with identical sorted strings are candidates for parallelization. This doesn't do precisely the same thing theoretically as finding subgraphs; it is still correct and a great deal faster, but it's overly conservative. We are also able to coalesce strings as vectors, augmenting GCC to annotate instructions with their colors as well as the colors of other instructions they could be combined with. However, we were unable to automatically complete the instruction vectorization, due to time constraints.

6.2 Loop Unrolling

We developed support to try vectorizing code in three phases. First, on the original loop. Next, by unrolling the loop once (doubling the instructions in the loop) and trying the optimization, and then by unrolling the new loop once more (yielding four times the original loop) and trying the optimization.

The problems with loop-unrolling are two-fold.

First, loop unrolling only introduces new temporary registers, not ones assigned to scalars. This means that:

```
for (i=1 to 10); x=x+1
```

expands to

```
for (i=1 to 10 step 2); x=x+1; x=x+1
```

Unfortunately, the two additions of x are not vectorizable since they would have to reside in the same position in the vector.

Second, loop unrolling becomes very useful for our purpose when accessing arrays. Ideally, a loop would be unrolled up to four times and all four iterations would execute in parallel using vector operations. The problem here is that there is a dependency when accessing arrays on the loop induction varaiable (i.e. the loop counter) and the base address of the array.

Even though the base address is trivial to deal with because it is constant, our optimization constraints need to be relaxed to allow us to refer to the separate array iterations as distinct graphs even though they have a dependency on the (changing) induction variable.

6.3 Foreign Instruction Set Precludes Further Optimizations

One shortcoming of the implementation is that GCC 3.0.3 does not natively support the parallel instructions of the Emotion Engine, nor does it have good handling of 128 bit registers. It will, for example, allow you to define 64 or 128 bit registers but will perform loads/stores 32 bits at a time, combining the result [4]

[4] This is particularly true of GCC that targets Linux for the PlayStation 2. The version used by professional game developers handles 64 bit data elements a bit better).

A thorough implementation would have required making GCC fully aware of the large register sizes as well as introducing the parallel instructions. Since we did not have enough time for such an extensive update to GCC, we were forced to implement our optimizations by emitting inline assembly instructions that contained the parallel instructions. Because inline assembly instructions are treated as somewhat of a black box by GCC, this blocks some of the other optimizations GCC is capable of after performing it's loop optimizations. The assembler, however, is still capable of doing a little bit of instruction reordering because it processes all assembly instructions.

6.4 Vectorization and De-vectorization

In our implementation, we vectorized pairs of registers by using the *PCPYLD* (Parallel Copy Lower DoubleWord) instruction. This instruction takes the lower 64 bits of two registers and combines them into a single 128 bit value. We extracted the values by using the PEXEW (Parallel Exchange Even Word) instruction which swapped bits 0-31 with 64-95. This allows us to join two expressions:

`A=B+C`

`D=E+f`

as such:

`PCPYLD E, B, E`

`PCPYLD F, C, F`

(E now contains [B, E] and F contains [C, F]

`PADDW D, E, F`

(D now contains [B+C, E+F])

`PEXEW A, D`

(D now contains [B+C, E+F] and A contains [E+F, B+C])

Even though D and A contain both elements in the vector, all of the operations in GCC will only operate on the lower 32 bits of the registers so the remaining elements are ignored.

7 Future Work

7.1 More Loop Unrolling

An interesting next step would be to attempt to formalize and implement the unrolling optimization, which was briefly mentioned earlier. This optimization is meant to create opportunities for the optimization we have described. If there are not many vectorizable variables in a loop body, then unrolling a loop a few times will create several definitions of the variable which may be parallelized together, provided the operation defining the variables

is well-behaved. More precisely, the operation should be associative, commutative, and have an identity.

Here's an example of how the optimization might work. Consider the loop:

```
for i = 1 to n
sum += sum + a[i]
end for
```

Assume for simplicity that n is a multiple of 4. Unrolling four steps gives us:

```
for i = 1 to n step 4
sum += sum + a[i]
sum += sum + a[i+1]
sum += sum + a[i+2]
sum += sum + a[i+3]
end for
```

The optimization would first renumber variables in order to parallelize:

```
sum1 = sum; sum2 = sum3 = sum4 = 0
for i = 1 to n step 4
sum1 = sum1 + a[i]
sum2 = sum2 + a[i+1]
sum3 = sum3 + a[i+2]
sum4 = sum4 + a[i+3]
end for
sum = sum1 + sum2
sum += sum3 + sum4
```

Then parallel instructions would be inserted:

```
v = (sum, 0, 0, 0)
for i = 1 to n step 4
w = (a[i], a[i+1], a[i+2], a[i+3])
v = padd(v,w)
end for
sum = v[0]+v[1]
sum += v[2]+v[3]
```

Observe that the above transformation may be applied to *any* variable defined with an operation that is associative, commutative and has an identity. (For multiplication, we would have the extra variables initialized to 1 instead of 0.) The catch is that it seems difficult to cleanly formalize interesting conditions for the above optimization. The following is a particularly nasty example, pertaining to array alignment:

```
for i=n to m; a[i]=a[i]+1
```

In the worst case, $n \bmod 4 \neq 0$ and $m \bmod 4 \neq 0$; here, we'd only be able to vectorize the some portion of the loop. In the unrolling, we have to run up to three iterations before the loop and up to three iterations after the loop to ensure proper alignment.

7.2 Element Shuffling

There are a number of instructions in the Emotion Engine's instruction set that allows elements to be shuffled within 128 bit parallel registers. At the cost of a much higher complexity algorithm, this could be exploited to enable more complex vectorizations where vectorizable subsets of registers are available but their positions within the vectors need to change.

```
A=A1+A2
B=B1+B2
C=A1+B2
V1=[A1, B1]
V2=[A2, B2]
V3=PADD(V1,V2)
V2=EXCH(V2)
V4=PADD(V1,V2)
```

By the end of this code, V3 contains [A, B] and V4 contains [C, garbage]

7.3 Scalar Operations on Vectors

Because the Emotion Engine does not distinguish between vector registers and scalar registers, the 32 bit instruction set is still available for use with registers containing vectors. This would enable us to perform any operation with the vector element that is stored in the lowest 32 bits of the register. In future work, this would enable us to include the loop counter induction variable in a vector (if it is beneficial) because the loop exit test can still be performed on it. This same reasoning applies to any variable that needs to be treated separately and can be assigned a slot in the lowest 32 bits (or swapped using one of the shuffling instructions when needed), so long as the register is not being written to (which would eliminate the upper 96 bits).

References

[1] A. Bik, et al. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems. Intel Technology Journal, 1Q 2001.

[2] D. DeVries. A Vectorizing SUIF Compiler: Implementation and Performance. Master's Project, Dept. of Electrical and Computer Engineering, University of Toronto, 1997.

[3] D. DeVries and C.G. Lee. A Vectorizing SUIF Compiler. In Proceedings of the First SUIF Compiler Workshop, pp. 59-67, 1996.

[4] E. Hogan, G. Judd, and S. Sinnamohideen. Automatically Identifying Opportunities for Using Special Purpose Instructions. 15-740 Course Project, Carnegie Mellon University.

[5] IA-32 Intel Architecture Software Developer's Manual, Intel Corporation, 2003.

[6] C. G. Lee and M. G. Stoodley. Simple Vector Microprocessors for Multimedia Applications, pp. 25-46, MICRO 1998.

[7] D. Naishlos et al. Compiler Vectorization Techniques for a Disjoint SIMD Architecture. IBM Research Report, November 2002.

[8] EE Core Instruction Set Manual Version 5.0, Sony Computer Entertainment Inc., 2001.

[9] G. Valiente. Simple and Efficient Tree Matching. Departament de Llenguatges i Sistemes Informtics, Research Report Number LSI-00-72-R, 2001.