Architecture-Independent Meta-Optimization by Aggressive Tail Splitting

Michael Rock and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 23, D-38106 Braunschweig, Germany
`rock,koch@eis.cs.tu-bs.de`

Abstract. Several optimization techniques are hindered by uncertainties about the control flow in a program, which can generally not be determined by static methods at compile time. We present a novel approach that aims to alleviate this limitation by explicitly enumerating all control paths through a loop nest, thus allowing wider and more efficient use of standard optimization passes later. Our analysis will address the possible explosion in code size, which despite high theoretical upper bounds appears to be acceptable in practice, as well as performance gains both for RISC and CISC target processors.

1 Introduction

The quality of many optimization algorithms is directly dependent on the amount and accuracy of the information retrieved by prior analysis steps. Despite continuous progress in the development of such tools, certain program structures invalidate the analysis efforts: One of these cases is the joining of two or more different flows of execution after a conditional at a so-called merge point in the control flow graph. At such a point, the analysis is uncertain which of the multiple paths was actually taken to arrive at the merge point. Thus, it cannot propagate beyond the merge point any assertions (e.g., on variable values and memory contents) that are not common to all of the execution paths arriving at the merge point.

2 Related Work

The criticality of analysis across merge points has been recognized previously. One technique that is already practically used in many compilers (e.g., using **`-ftrace`** for GCC [1]) is tail duplication [2]. This approach is used for the joining of blocks into single-entry point hyperblocks by copying a dedicated instance of all shared tail blocks into each individual hyperblock, thus reducing the number of merge points. The technique is also commonly used for improving the scheduling of the generated code [3] [4]. Some research has been done in combining loop unrolling and tail duplication [5], but the algorithm presented there does not re-splice the program as done in our approach.

3 Novel Approach

However, these existing approaches (both for hyperblock construction and scheduling) limit the scope of their operation to limited areas such as only within loop bodies. The algorithm presented here has all the advantages of conventional tail splitting and loop unrolling, but now the back edges are re-spliced between specialized versions of the loop body and do not lead back to the initial loop header. This unrolling, splitting and re-splicing operation was termed *Aggressive Tail Splitting* (ATS). Its effect is to keep disjunct paths of execution isolated for as long as feasible, thus avoiding the creation of analysis-impeding merge points. The approach was inspired by the basic use of the tail splitting transformation in an interpreter [6].

4 ATS Algorithm

We will introduce the algorithm using the trivial example of the C source code fragment shown in Figure 1.

```
TYPE b0(TYPE *a) {
    TYPE s=0;
    int i;
    for (i=1; i < SIZE; i++) {
        if (a[i] < 0) {
            a[i] = 0 ;
        }
        s += a[i] * a[i-1] ;
    }
    return s;
}
```

Figure 1. Initial input source code

Since some of our later analyses of ATS effects will be performed with disabled compiler optimizations (to focus on ATS itself), we will perform some simple optimizations such as Common Subexpression Elimination (CSE) manually on **`b0()`**, leading to the version **`b1()`** shown in Figure 2.

```
TYPE b1(TYPE *a) {
    TYPE s=0;
    int i;
    for (i=1; i < SIZE; i++) {
        // Block 1
        TYPE t=a[i] ;   // perform CSE manually ...
        if (t < 0) {    // ... here ...
            // Block 2
            a[i] = 0 ;
            t = 0 ;     // ... here ..
        }
        // Block 3
        s += t * a[i-1] ;// ... and here
    }
    return s;
}
```

```
TYPE b2(TYPE *a) {
    TYPE s=0;
    int i;
    for (i=1; i < SIZE; i++) {
        TYPE t = a[i] ;
        if (t < 0) {
            a[i] = 0 ;      // tail split here ...
        } else {
            s += t*a[i-1] ; // ... and here
        }
    }
    return s;
}
```

Figure 2. Result of single intra-loop iteration tail splitting

4.1 Tail Splitting the Back Edge

We now begin by splitting all edges arriving at blocks with multiple predecessors, copying the original block backward along each of the split edges (see Figure 3). Initially, this occurs only within the loop body (as suggested in [6]), leading to the source code **`b2()`** shown in Figure 2.

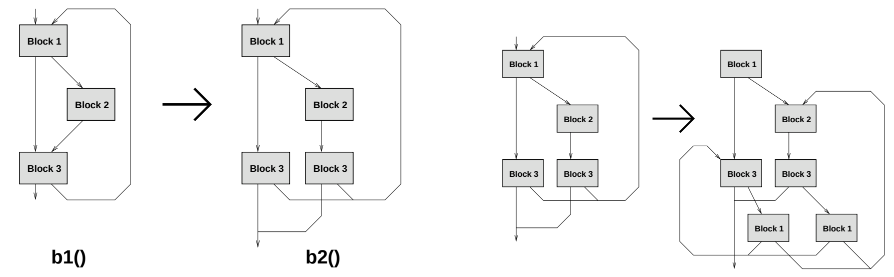

Figure 3. First inter-iteration tail-splitting and beginning of head duplication

At this stage, we have not yet achieved our aim of reducing the number of merge points. On the contrary, in **`b1()`**, Block 1 only had two arriving execution paths, while in **`b2()`**, it now has three arriving paths. To avoid cluttering this figure, an *arriving* edge can carry more than one path of execution, the actual number is determined by the number of *originating* nodes.

Beyond conventional tail-splitting, ATS now *continues* this process along the back-edges, thus crossing loop iteration boundaries. This step is shown in Figure 3: The two back edges have been tail split and replaced by copies of the loop header Block 1. As before, however, the number of execution paths arriving at merge points has increased. Blocks 2 and 3 now both have three arriving paths. Due to the increasing convolution of the structure at this point, we will no longer show the source code of these intermediate steps.

Undeterred by the temporary decrease in solution quality, ATS continues in the same fashion. Later on, it generates the structure shown in Figure 4.a: Now, three distinct copies exist of the loop body. The entire first iteration has been peeled off and serves as a dispatcher to the specialized loops lower in the hierarchy. At this stage of the process, a specialized loop exists for each execution path arriving there. In the example, the loop B was created for the case that the condition in **`b2()`** was true during the previous (peeled-off) iteration, and the loop A for the inverse case.

4.2 Termination Conditions for the Tail Splitting Process

At this stage in the discussion, we have to address the question of when to stop splitting edges. Even with the structure shown in Figure 4.a, we could continue the process. It is thus useful to step back from the algorithm itself and re-examine the aim of our methods.

The main goal of ATS is to enable more aggressive optimizations inside of loop bodies. The manner in which this should be achieved is by providing other optimization passes with accurate information (assertions) about the execution path that lead to a

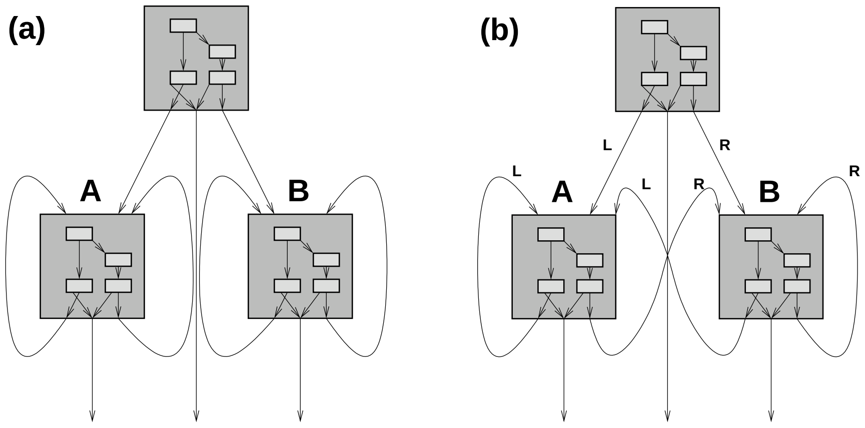

Figure 4. (a) Loop unrolling and tail spliting, (b) ATS

given node (history). Based on these assertions, these optimizations can then perform their own transformations or scheduling decisions.

Thus, to arrive at the termination condition of the tail-splitting phase, we need to determine the minimal required length of the history. It turns out that for loops, this length depends on the distance vectors for loop-carried dependencies. In our example, **`b0()`** has a distance vector of $d : \{0,1\}$, indicating that a calculation at **`a[i]`** depends on **`a[i-0]`** and **`a[i-1]`**. This implies that in this case, there is no need to keep track of data or control flow beyond the previous loop (the one that computed **`a[i-1]`**). Thus, the length of the history required is the maximum range of the distance vectors for loop carried dependencies.

By closely examining the digraph we use to represent the program structure (Figure 3 and 4.a), it can be determined that during the tail-splitting phase, this digraph will always have the same tree-like nature: The root and inner nodes serve as dispatcher to specialized loops represented by the leaf nodes (here A and B), with the only back-edges occurring at the leaf-level.

It is thus obvious that the depth of the digraph (the longest path from the root to a node) is limited by the length of the history: Each level of the graph represents a loop-carried dependency on a previous iteration.

4.3 Inter-Leaf Node Re-Splicing

Thus far, our approach relied on repeatedly applying a well-known technique to transform the program. But now that we have determined when to terminate this process, we can transform the leaf-level edges to accurately encode the histories without further expansion of the digraph. This is achieved by connecting each of the back-edges leaving a leaf-level node to the leaf-level node (same or different) that was specialized for that specific intra-loop execution path.

The method will become clearer when illustrated using the example in Figure 4.a. As stated previously, there are only two history-specific back-edges out of the loop bodies[1], depending on the condition $P \equiv$ **`if (t<0) ....`**

For the example, we will thus ensure that all paths R which had $P =$ true during the previous iteration will lead to the loop specialized for this case, namely B. Analogously, all paths L that had $P =$ false in the previous iteration will be processed in the specialized loop A. The result of this transformation is shown in Figure 4.b. The histories leading to a loop body are thus encoded in the selection of which specific specialized leaf-level loop is executed. As a result, a later optimizer processing the loop bodies A or B can now be certain which assertions can be exploited. For brevity, the actual source code for this structure, termed **`b3()`** in the following text, is omitted here.

5 Performance

To evaluate the performance of code subjected to ATS, we compiled the the processing stages **`b1()`** to **`b3()`** of our example onto the UltraSPARC III+ and Athlon XP targets. For the SPARC, we used the Sun Workshop C Compiler 5.3 with maximum optimization effort (option **`-fast`**). On the Athlon, we employed GNU GCC 3.2 with the **`-O3`** option. The results are summarized in Table 1.

Version		SPARC		Athlon	
		Time [s]	Speedup	Time [s]	Speedup
plain	b1	6.71	1.00	3.07	1.00
tail-split	b2	6.55	1.02	2.53	1.21
ATS	b3	5.90	1.13	1.95	1.57

Table 1. Performance for the example code

So even for this trivial example, some modest performance gain (up to 57% on the Athlon) was realized. Before analyzing a larger, more meaningful application in Section 5.2, we will consider the effects of ATS on later optimization passes.

5.1 Increased Optimization Opportunities

As seen in Figure 4.b, even the post-ATS version **`b3()`** contains hyperblocks with multiple predecessors. However, now all paths even through different predecessors keep track of the same history (and thus assertion sets), increasing the opportunities for standard optimization techniques. Thus, constant propagation or CSE can now be applied across iteration boundaries. This is not possible when the loop is not tail split because of the uncertainty of the execution path leading to a block. Another technique that allows such inter-iteration optimization is loop unrolling. However, even here the path uncertainty problem remains and limits the applicability of the standard optimizers.

[1] Since the center edge is not a back-edge (loop exit), it is disregarded here.

5.2 Effect of ATS on Control Flow

To evaluate the effect of ATS on a more meaningful program, we now consider a decoder for fixed-table Huffman codes. Since this algorithm actually contains a substantial control-content in its loops (in contrast to many DSP codes), it is a good candidate for ATS. Due to the size of the intermediate and final stages, we will only list the original source program in Figure 5.

```
void huffDec(char* src, char* dst, int limit) {
    int i;
    int pos = 1<<7 ;
    for (i=0; i < limit; i++) {
        int index =0;
        int max ;
        /* Maximum bit length*/
        for (max=0; max < 6; max++) {
            if (*src & pos) {
                index++;
            }
            pos >>= 1 ;
            if (!pos) {
                pos = 1<<7 ;
                src++ ;
            }
            index = table[index] ;
            if (index & 1) {
                *dst++ = index>>1 ;
                break ; /* ready, early exit*/
            }
        }
    }
}
```

Figure 5. Huffman decoder source

As before, the code is evaluated on the SPARC and Athlon targets. For each target, Table 2 shows three versions: The original, a fully-expanded ATS variant, and finally an optimized ATS version. The latter was subjected to agressive propagation, replacing constant accesses to constant memory locations with their contents. In both ATS cases, the inner loop has been completely unrolled to remove all internal merge points. The size given is the size of the compiler generated object code (`.text` segment).

The Huffman decoder has a theoretical upper bound of growth [2] in the number of instructions by a factor of 299593. However, the maximum observed growth factor of the optimized ATS digraph is only 56. More aggressive compiler optimization options (`-fast` and `-O3`) exploit the ATS'ed structure even further, leading to a maximum growth of only 16.01. This is well within the practical limits established by other techniques such as aggressive loop unrolling (e.g., by a factor of 96 in [7]).

In terms of performance gain, on the SPARC target, at best 52% were achieved (interestingly not at the highest compiler optimization level). On the Athlon however, the highest speedup was over 350%.

When examining the final assembly code, it becomes apparent that its post-ATS structure is completely different from the original source code: Now, the entire fixed Huffman table has been inlined into the program, resulting in an actual FSM for decoding the bit stream. This was possible due to the fact that the *index* variable now was subject to constant propagation, making the value constant for all reads. This reduces

[2] Due to space limitations, the relevant analyses cannot be shown here.

Compile	Stage	Size [B]	Time [s]	Growth	Speedup
SunCC	base code	164	6.08	1.00	1.00
-O1	complete ATS	480	9.65	2.93	0.63
SPARC	optimized ATS	1300	3.99	7.93	1.52
SunCC	base code	168	2.79	1.00	1.00
-fast	complete ATS	1600	6.1	9.52	0.45
SPARC	optimized ATS	1560	2.47	3.29	1.13
gcc	base code	103	1.78	1.00	1.00
-O1	complete ATS	193867	1.46	1882.20	1.23
Athlon	optimized ATS	5795	0.93	56.26	1.91
gcc	base code	109	1.59	1.00	1.00
-O3	complete ATS	30052	0.69	275.70	2.30
Athlon	optimized ATS	1745	0.45	16.01	3.53

Table 2. Results of the HuffDecode benchmark on Sparc and Athlon, time and growth

the number of memory accesses considerably, since the table walk is now encoded into the program's control flow. By lowering the pressure on the memory, more ILP can be exploited.

6 Interpreting the Benchmark Results

Since ATS itself is no optimization, the origin of the performance gains should be examined. In the initial example (**`b1()`**...**`b3()`**), the speed-up was realized by removing multiplications by zero after constant propagation.

The Huffman decoder of Figure 5 is more complex. A GCC-compiled version was analyzed in greater detail using processor-internal performance counters on both platforms. These measurements are shown in Table 3. Of interest are the actual number of instructions executed and the number of cycles stalled due to branch mispredictions.

Stage	Instructions [10^9]	Mispredicts [10^6 Cycles]	Rel. Insts	Rel. Stalls
base code SPARC	3.85	12.11	1.00	1.00
complete ATS SPARC	3.58	72.25	0.93	5.96
optimized ATS SPARC	1.50	50.29	0.39	4.15
base code Athlon	5.04	73.04	1.00	1.00
complete ATS Athlon	2.30	63.95	0.46	1.03
optimized ATS Athlon	2.00	58.17	0.37	0.93

Table 3. Results of the HuffDecode benchmark on Sparc and Athlon-XP, instructions and stalls

Since the ATS operation imposes a more complex control structure on the program (multiple value-specialized versions for each execution path), the quality of the processor's branch prediction logic becomes crucial. In the numbers shown above, it is obvious that the unit in the SPARC is not able to cope with the increase in complexity (more mispredict cycles) of the complete ATS form. The Athlon's predictor, however, can easily handle that version (only minimal increase in mispredict cycles).

The performance gains of ATS are mainly due to the reduced number of instructions actually executed (see Table 3). E.g., for the Athlon, only 30% of the original number of instructions is executed after ATS. This reduction is achieved by giving the later optimization stages (in this example: especially the constant propagation step) a larger set of assertions to work with. Since the constant propagation in the 'optimized' versions extends to removing references to the constant Huffman table, 71.9% of all memory accesses are eliminated in the post-ATS code (on both platforms).

7 Future Work

We are currently integrating an automatic ATS pass into a complete compile flow, thus allowing a more thorough experimental evaluation in the future. Furthermore, the ATS digraph itself is amenable to high-level minimization techniques which we expect to limit the practical code growth even further.

8 Conclusion

The ATS transformation can considerably increase the opportunities for applying standard optimization techniques to inner loops. In practice, the excessive theoretical growth in code size can often be avoided. By selectively applying the transformation, the growth penalty need only be accepted when economical (e.g., in the range of 100-200 and only for time-critical inner loops).

References

1. Cohn, R., Lowney P.G., "Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha", *Journal of Instruction-Level Parallelism*, May 2000
2. Gregg, D., "Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling", in *Proceedings of the 9th International Conference on Compiler Construction (CC 2001)*, pp. 200-212, LNCS 2027, Genoa, April 2001
3. Gao, G.R., Amaral, J.N., Dehnert, J. , Towle, R., "The SGI Pro64 Compiler Infrastructure", *Intl Conference on Parallel Architectures and Compilation Techniques (PACT)*, tutorial, 2000
4. Huiyang, Z., Jennings, M.D., Conte, T.M. "Tree Traversal Scheduling: A Global Technique for VLIW/EPIC Processors", *Proc. 14th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, LNCS, Springer Verlag, August 2001
5. Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G, Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. "The superblock: an effective technique for VLIW and superscalar compilation." *The Journal of Supercomputing 7*, 1/2 (May 1993), 229-248.
6. Bala, V., Duesterwald, E., Banerjia, S., "Dynamo: A Transparent Dynamic Optimization System", *ACM SIG-PLAN Notices*, vol. 35, pp. 1-12, 2000.
7. Lowney, P.G., Freudenberger, S.M., Karzes, T.J., "The Multiflow Trace Scheduling Compiler", *The Journal of Supercomputing*, vol. 7, number 1-2, pp. 51-142, 1993