Optimal Huffman Tree-Height Reduction for Instruction-Level Parallelism

Technical Report TR-08-34
Department of Computer Sciences
The University of Texas at Austin

Katherine Coons COONSKE@CS.UTEXAS.EDU
Warren Hunt WHUNT@CS.UTEXAS.EDU
Bertrand A. Maher BMAHER@CS.UTEXAS.EDU
Doug Burger DBURGER@CS.UTEXAS.EDU
Kathryn S. McKinley MCKINLEY@CS.UTEXAS.EDU

Abstract

Exposing and exploiting instruction-level parallelism (ILP) is a key component of high performance for modern processors. For example, wide-issue superscalar, VLIW, and dataflow processors only attain high performance when they execute nearby instructions in parallel. This paper shows how to use and modify the Huffman coding tree weight minimization algorithm to expose ILP. We apply Huffman to two problems: (1) *tree height reduction*–rewriting expression trees of commutative and associative operations to minimize tree height and expose ILP; and (2) *software fanout*–generating software fanout trees to forward values to multiple consumers in a dataflow ISA. Huffman yields two improvements over prior work on tree height reduction: (1) it produces globally optimal trees even when expressions store intermediate values; and (2) it groups and folds constants. For fanout, we weigh the targets by the length of the critical path from the target to the end of its block. Given perfect weights, the compiler can minimize the latency of the tree using Hartley and Casavant's modification to the Huffman algorithm. Experimental results show that these algorithms have practical benefits, providing modest but interesting improvements over prior work for exposing ILP.

1. Introduction

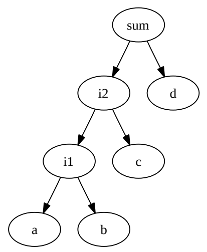
Increasing instruction level parallelism (ILP) improves performance when executed on multi-issue processors. A well-known technique to increase ILP is *tree height reduction*, in which the compiler exploits commutativity and associativity to transform expression trees such that their height is minimized [1, 2, 3, 4]. A related problem is how to construct a *fanout tree* that replicates a producer's output value for all consumers. This problem arises in dataflow machines when the compiler needs to generate a software fanout tree with the goal of minimizing the latency on the critical path and maximizing ILP.

This paper applies the Huffman coding algorithm to these problems to guarantee a minimum average path length for a given tree. Previous work by Baer and Bovet [1] gives an algorithm for constructing arithmetic trees, and Beatty proves that their approach minimizes tree height [2]. For simple expressions that consist of only one type of operator and have operands (leaves) of equal weight, Huffman coding produces a balanced binary tree (see Figure 1), which is the same tree produced by Baer and Bovet [1]. However, when leaves have unequal weights, Huffman produces an unbalanced tree that minimizes the weighted sum of the heights rather than the maximum height. This property allows the Huffman algorithm to globally balance a tree with leaves that are themselves subtrees. In a compiler, such a tree occurs when an expression computes intermediate values that must be preserved. To minimize the total expression height, the compiler applies Huffman first to the subtrees rooted at the intermediate values, and then optimize the global tree using the weights of the subtrees. With a simple extension, the Huffman algorithm can also expose opportunities for constant folding. If the compiler initializes the weight of constants to zero, the Huffman algorithm will combine and fold them. These optimizations result in a globally optimal tree with preserved intermediate results. This approach naturally handles trees of mixed-precedence operators similarly to previous work [1, 3].

We also apply Huffman to the problem of minimizing delay due to fanout instructions in a dataflow ISA. This problem is very close to minimizing tree height for synchronous hardware circuit layout, which Hartley and Casavant solved optimally [5]. Their algorithm follows essentially the same steps as Huffman's, except that it optimizes the length of the longest path through the tree instead of the sum of the weights at each step. If all the nodes are of weight one, both algorithms produce the same tree height. With non-unit weights, Hartley and Casavant produce a provably minimal tree height [5]. Huffman, however, minimizes the weighted sum of the path lengths, and thus the maximum height can be longer. We want to build a fanout tree that minimizes the

```
i1 = a + b
i2 = i1 + c
sum = i2 + d
avg = sum / 4
```

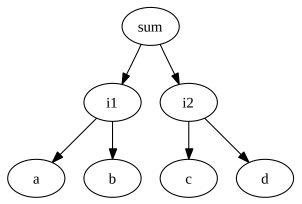
(a) Original



(b) Original tree $sum = ((a + b) + c) + d$

```
i1 = a + b
i2 = c + d
sum = i1 + i2
avg = sum / 4
```

(c) After reduction



(d) After reduction

Figure 1: Example code and expression trees before and after tree height reduction.

critical path through the basic block (really, the program). We compare three approaches: (1) a balanced tree using a weight of one for each node; (2) a Huffman tree with targets weighted by critical path length; and (3) a Hartley and Casavant tree with targets weighted by critical path length. To estimate each target's importance, we weigh nodes by the path length from the target operand to the bottom of the graph. If these estimates were perfect, Hartley and Casavant would always minimize the critical path length. Since there are dynamic effects that the compiler cannot perfectly estimate, we find that Hartley and Casavant's algorithm performs better in most cases, but performs poorly in some cases where the criticality estimates are inaccurate. The Huffman algorithm, in contrast, provides more modest improvements when the estimates are accurate but also tolerates inaccuracies more easily. We believe these are the first published software fanout algorithms for dataflow.

We implement tree height reduction and fanout insertion in a C and Fortran compiler, and simulate performance of a hybrid dataflow microarchitecture [6, 7] running the EEMBC benchmarks and kernels extracted from SPEC2000 and elsewhere. The simulator is validated to within 5% of the RTL for the hybrid dataflow microarchitecture. For fanout insertion, both Huffman and Hartley and Casavant improve over Balanced on EEMBC by an average of 4.3% and 5.2%, respectively. Occasionally, Huffman outperforms Hartley and Casavant and in general has lower variation in performance. For arithmetic tree height reduction, Huffman balancing provides no performance benefit over Baer-Bovet. While arithmetic tree height reduction improves microbenchmark performance by up to 17%, we find few opportunities in the EEMBC benchmarks, suggesting that typical programs may not benefit from the optimization.

This paper demonstrates that using Huffman as a basis for maximizing instruction level parallelism offers some modest theoretical and practical advantages.

2. Related Work

Baer and Bovet introduced tree height reduction to expose ILP in associative and commutative expression trees, assuming operations and references have the same cost [1]. Their algorithm takes multiple passes, rewriting expressions at each level of the resulting tree. Intuitively, it produces the shortest tree, and Beatty proves that this tree has minimal height [2]. The Huffman formulation offers two advantages over Baer and Bovet: it minimizes expressions that must preserve intermediate values, and it exposes opportunities for constant folding. For complex expressions, the Huffman algorithm produces trees that expose more ILP than those produced by the Baer and Bovet algorithm.

Kuck presents a similar algorithm that includes distributivity, but is not provably minimal, and poses a harder problem: minimizing tree height and resource requirements (e.g., minimizing the number of adders in use simultaneously) [3]. For example, Figure 1(b) uses only a single adder at any given time, and could be implemented with a single register, whereas the reduced tree in Figure 1(d) requires two registers and two adders. Landwehr and Marwedel describe a genetic algorithm to reduce both for hardware layout [8]. If we just consider the number of registers in use, Sethi-Ullman numbering minimizes the number of registers used by an arithmetic expression [9]. This algorithm was designed for in-order architectures with small register files, and decreases ILP.

More recently, Mahlke et al. [4] evaluate Baer and Bovet's algorithm and show that even though it increases register pressure, it improves performance on a VLIW machine and they recommend it for use in VLIW compilers. Since Huffman strictly increases the ILP exposed when compared to Baer and Bovet, it is equally applicable to VLIW.

Huffman introduced an algorithm for encoding messages based on their expected frequencies that builds a coding tree that provably minimizes the global sum of the leaves [10]. Given leaves each with weight w_j that can be arbitrarily reordered, the algorithm first sorts the leaves and places them in a worklist. It then repeatedly takes the two lowest cost elements out of the worklist, combines them into a subtree, weights the interior node by the sum of the elements, and inserts the result in the appropriate place in the sorted list. It repeats this process until the worklist is a single element. This algorithm also solves the tree height reduction problem for expressions consisting of multiple operations of a unique commutative and associative operator (e.g., an expression consisting entirely of additions of operands—leaves—with unit weight). We extend this algorithm to handle arbitrary expressions with mixed precedence and operators, storage of intermediate values, and constant folding.

We also apply Huffman to build software fanout trees for a dataflow ISA. The prior hardware and compiler dataflow literature mentions that they require the compiler to insert fanout instructions but, to our knowledge, does not describe or compare algorithms for this task [11, 12, 7, 13].

Hartley and Casavant rearrange arithmetic expressions to construct adder circuitry using a Huffman-like algorithm [14]. Their optimization goal is to minimize delay through the circuit and *shim* (storage circuits for ready but not yet required operands) by minimizing the height of a weighted tree. They weigh the leaf operands by their *delay*, i.e., the known cycle at which they will be available. Hartley and Casavant's algorithm is the same as Huffman's algorithm, except that when it combines two nodes, it weighs the resulting node by the maximum of the children's weights plus one, rather than the sum of the children's weights. Because the delay is known and fixed, this algorithm provably minimizes the critical path length through the resulting circuit. We cast the problem of building a software fanout tree in a similar way, by estimating the output latency of each target instruction. The compiler has only static estimates, however, whereas the input latencies are known in Hartley and Casvant's experiments. We compare Hartley and Casavant's algorithm to Huffman's algorithm for software fanout tree construction, which minimizes the total weight [10, 15].

3. Tree Height Reduction

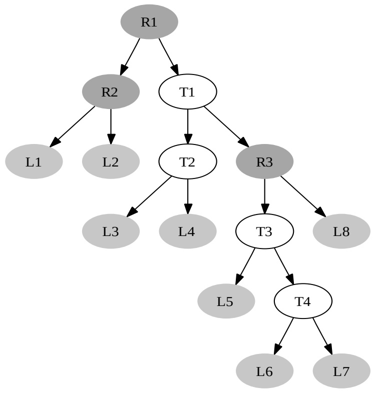
This section first motivates using Huffman's algorithm for tree height reduction, and then describes our intermediate form and how we find candidate expressions. Next, we introduce Huffman tree height reduction and show how it handles intermediate expressions and constants. We compare the resulting trees from Huffman to Baer and Bovet on two examples. For simplicity of exposition, we restrict our discussion to associative and commutative operations. However, we handle subtraction as an addition of a negative and other operations in the usual way [1].

The example provided in Figure 2(a) motivates using Huffman for tree height reduction. The nodes labeled L are leaves, the nodes labeled T are intermediate temporaries, and the nodes labeled R are retained results which must be preserved. This tree has three retained values, all of which would be considered roots in previous algorithms. For example, Baer and Bovet would balance each subtree independently and produce the optimized tree displayed in Figure 2(b). Notice that although each sub-tree is optimally balanced, the overall structure is not.

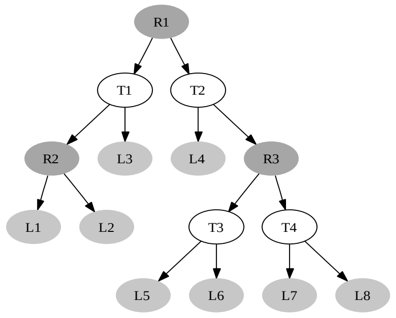
3.1 Intermediate Form

Tree height reduction operates on instructions in a basic block, where each instruction consists of a result operand (R), an operator (op), and two input operands (Ra, Rb). We use Static Single Assignment (SSA) [16], which gives each variable assignment a unique name, and renames all the uses of an assignment. When a use may get its definition from more than one assignment, SSA inserts ϕ functions, which specify a new name for a definition that selects from multiple definitions based on program control flow. Consider again the code in Figure 1(a). Intuitively, we see that values $i1$ and $i2$ are temporaries that are only used in the computation of sum. Because sum is used by a division instruction rather than another addition operation, it is the end of the chain of additions. The tree in Figure 1(b) depicts the computation of sum. The root of the tree is the value of sum, which is computed last in program order.

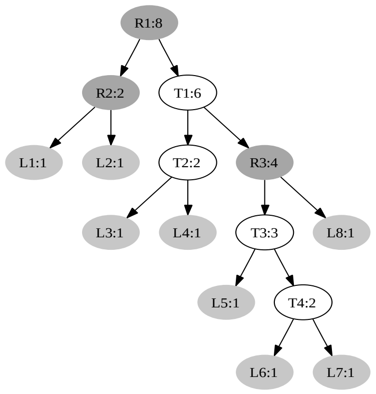
This intermediate representation slightly changes the problem presented by Baer and Bovet, who assumed expressions were already in a flattened form, such as sum = ((a+b)+c)+d), as shown in Figure 1. Therefore, we need an extra step to find candidate expressions in the basic block on



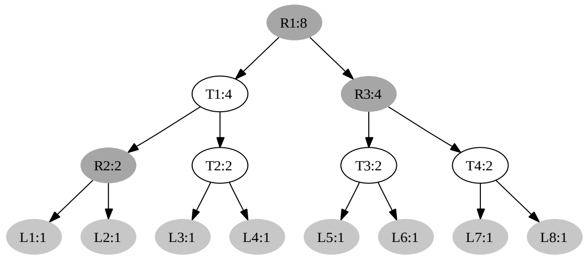
(a) Eight leaves & 3 retained values



(b) Baer and Bovet on retained values



(c) Weighted Tree: leaf weight = 1; interior nodes weight = $\sum$ child weight



(d) After Global Weighted Huffman coding

Figure 2: Example of globally optimal weighting

which to perform tree height reduction. We therefore proceed in two steps: (1) identify expression roots and temporaries, and (2) given a root of commutative and associative operations, construct an expression tree. The Huffman tree height reduction step keeps leaves and subexpressions in a list ordered by their weights (leaves have a weight of one and subexpressions have weight equal to the sum of their leaves). The algorithm combines the two lowest cost list elements, sums their weights, and puts the weighted subtree back in the list, until the list consists of a single element.

The pseudocode in Figure 3 implements this algorithm for our intermediate representation. The FindRoots function performs a forward pass that identifies root instructions. BalanceTree works backwards to find all the leaves associated with a given root, and then the *while* loop balances each tree.

3.2 Identifying Roots and Temporaries

To handle a RISC-like intermediate form (which many compilers now use), we identify roots, order them by precedence, and process higher precedence roots first. To distinguish roots and temporary definitions, we perform a forward pass over the instructions in a basic block. This step identifies variable definitions from associative and commutative operations as potential roots. If the root definition has a single use and both the definition and use statements have the same operation, then we classify the definition as a temporary instead of a root.

3.3 Huffman Tree Height Reduction

To minimize the sum of the depths of the leaves, the Huffman tree height reduction algorithm assigns each instruction a weight equal to the number of values that contribute to the result. Leaf operands having no sub-tree are given a weight of one. Figure 3 shows the algorithm first identifying all potential roots, and then ordering them by operator precedence. In precedence order, the Huffman algorithm then searches backwards from each root to collect the expression, putting the root's operands in the worklist. It chains up the use-def chains with (Def(Pop(worklist)). If the definition of the operand is itself a root (e.g., an intermediate value), the algorithm checks if its subexpression is balanced. If it is, it simply inserts it in weighted order in the list. If not, it balances the subexpression and then inserts it into the worklist with its appropriate weight. If a chained instruction is not a root and has the same operator as the root, its operands are simply added to the worklist. By handling higher precedence roots first, we obey correctness constraints, and any operation subtree of higher precedence is weighted correctly when we balance the lower precedence operation.

After the algorithm has collected the expression in the worklist (some subexpressions may be balanced already), it repeatedly pulls off the smallest two elements, creates an instruction to pair them, weights the result by their sum, and inserts the result into the sorted list.

```
// Find expression roots in a chain of associative and commutative operations
FindRoots()
  for each instruction I = 'R <- op, Ra, Rb'
    if op(I) not associative or commutative
      continue
    // I is a root unless R is a temporary
    //    (used more than once or by an instruction with a different operator)
    if NumUses(R) > 1 or op(Use(R)) != op(I)
      mark I as root, processed(root) = false
  order roots such that precedence of op(r$_i$) $\leq$ precedence of op(r$_{i+1}$)
  while roots not empty
    I = 'R <- op, Ra, Rb' =  Def(Pop(root))
    BalanceTree(I)

BalanceTree(root I)
  worklist: set
  leaves: vector

  mark I visited
  Push(worklist, Ra. Rb)
  // find all the leaves of the tree rooted at I
  while worklist not empty
    // look backwards following def-use from use
    T = 'R1 <- op1, Ra1, Rb1' = Def(Pop(worklist))
    if T is a root
      // balance computes weight in this case
      if T not visited
         BalanceTree(T)
      SortedInsert(leaves, T, Weight(T))
    else if op(T) == op(I)
      // add uses to worklist
      Push(worklist, Ra1, Rb1)

  // construct a balanced tree from leaves
  while size(leaves) > 1
    Ra1 = Dequeue(leaves)
    Rb1 = Dequeue(leaves)
    T = 'R1 <- op1, Ra1, Rb1'
    insert T before I
    Weight(R1) = Weight(Ra1) + Weight(Rb1)
    SortedInsert(leaves, R1, Weight(R1))
```

Figure 3: Pseudocode using the Huffman Optimization

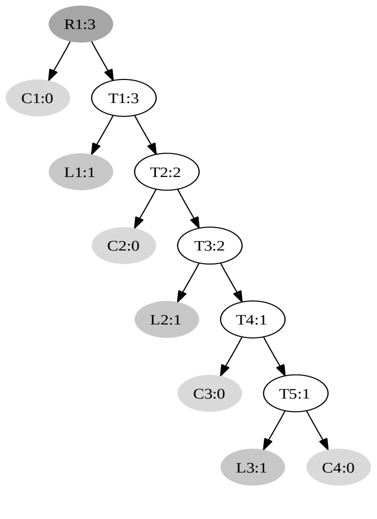
Huffman proves that this procedure produces trees with minimum $\sum_{leaf \in leaves} depth(leaf) \times weight(leaf)$ [10]. Because the weight of a leaf is equal to the number of values that contribute to its result this algorithm also minimizes the average depth of each leaf. Furthermore, by inserting new pairs into the sorted list after all other nodes of the same weight, this structure also provably minimizes the maximum depth (critical path length) of this expression [15].

Now consider again Figure 2(a). If we simply weigh the interior roots by their Huffman weight we obtain the weighted graph in Figure 2(c). Processing the roots in bottom up order, we first balance the sub-tree at R3, and then R2, resulting in a worklist of {L3:1, L4:1, R2:2, R3:4}. We then combine L3 and L4, yielding {T2:2, R2:2, R3:4}, then T2 and R2, yielding {T4:4, R3:4}, and finally the tree in Figure 2(d).

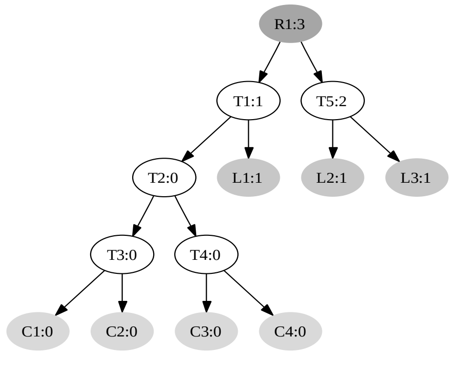
Notice that the same instruction can be the root of one tree and a leaf in another, and that each root and sub-tree may contain different associative operations. As a result, this formulation naturally handles retained interior nodes and mixtures of associative and commutative operations.

3.4 Constant Folding Optimization

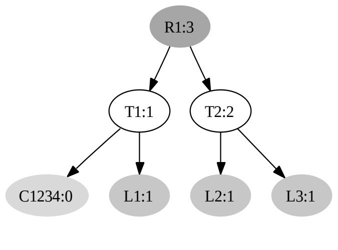
In addition to reducing the height of arithmetic trees, Huffman tree height reduction exposes opportunities for constant folding. Figure 4(a) shows an example tree containing several constants. Because the constants are paired with variables, the compiler is initially unable to perform constant folding. To enable the optimization, we assign all constants a weight of 0, which ensures that the algorithm pairs them first. This process produces a constant subtree of weight 0, as seen in Figure 4(b). The compiler can now evaluate and fold the constant expression to create Figure 4(c).



(a) Expression tree with constants



(b) Huffman with constant clustering

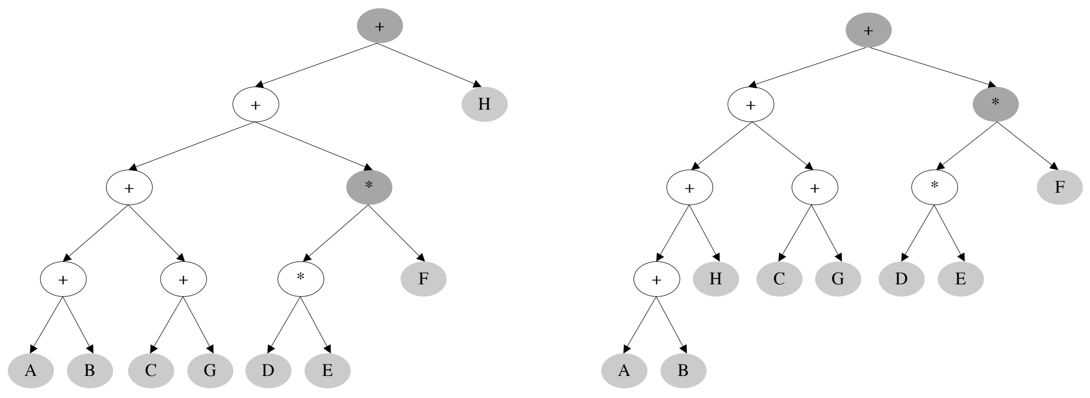


(c) Constants folded

Figure 4: Constant Clustering and Folding Example

3.5 Comparison to Baer and Bovet

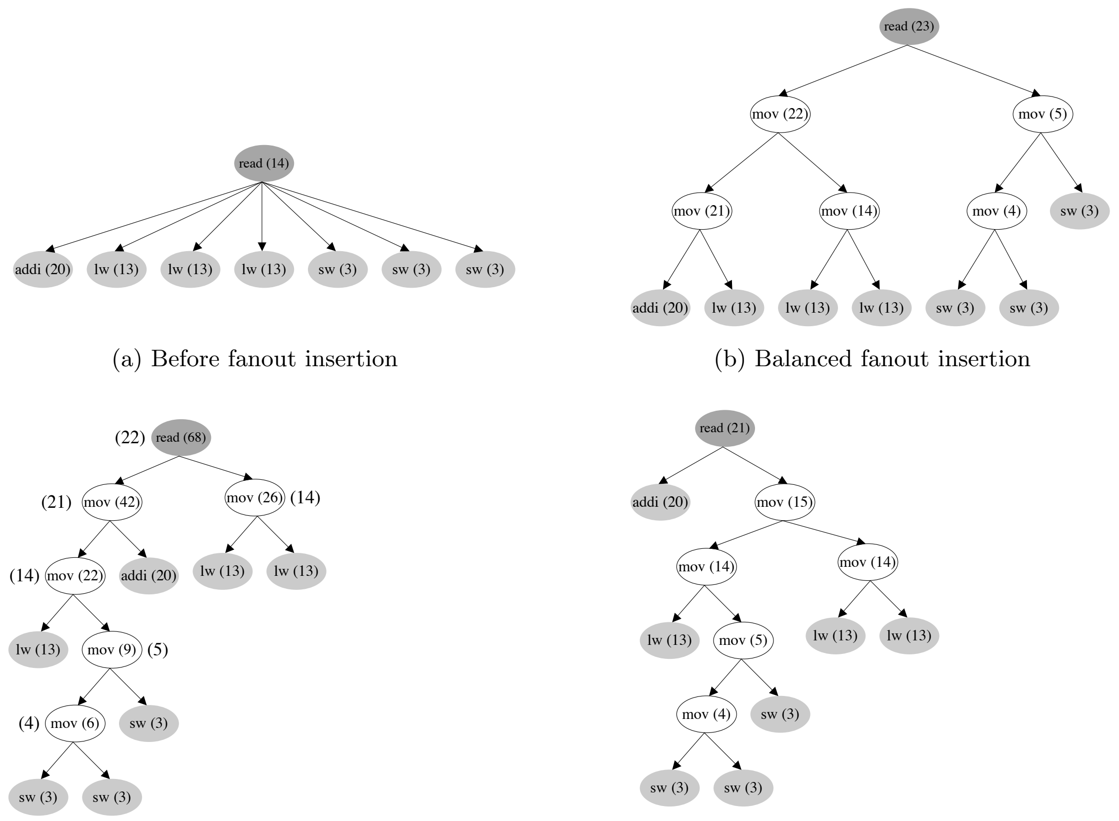
In their original formulation, Baer and Bovet process a flattened expression where each operand is a leaf, inserting each operand and operator in a worklist. Baer and Bovet's algorithm then takes multiple passes over the worklist. Each pass over the worklist collects pairs of same-precedence operations (obeying precedence constraints) and replaces the operands and operator with the resulting expression. If it cannot pair an operator and operand, the Baer and Bovet algorithm leaves them in the work list and continues to the next elements. It repeatedly processes the worklist in order until a pass performs no combining. We compare Baer and Bovet with Huffman using the example from their original paper [1]. Although both trees have the same maximum height, the sum of the heights in the tree produced by Baer and Bovet is larger. These trees perform equally well if all operands are available simultaneously, but if the arrival time of operands is non-uniform then the Huffman tree has a shorter average path length, and can thus tolerate delays to C, D, E, and G more easily. We show how to further exploit this insight in the next section.



(a) Baer and Bovet Version

(b) Huffman Version

Figure 5: Example: l = a + b + c + d * e * f + g + h



(a) Before fanout insertion

(b) Balanced fanout insertion

(c) Huffman (sum) fanout insertion

(d) Hartley and Casavant (max + 1) fanout insertion

Figure 6: Software fanout tree construction

.

4. Software fanout tree construction

This section describes the software fanout tree construction problem for dataflow ISAs and how to apply Huffman and Hartley and Casavant's algorithms to it. We begin by describing the architectural tradeoffs in the ISA for fanout.

Although an instruction may have an arbitrary number of consumers, dataflow ISAs typically impose a limit on the number of targets each instruction can encode. We refer to the number of consumers of a value produced by an instruction as the *fanout* of that instruction. To accommodate instructions with high fanout, the ISA may choose to provide variable width instructions that include a large number of targets, or it may choose a fixed instruction size with a limited number of targets and build a software fanout tree that explicitly forwards instructions.

An ISA where an instruction encodes a large number of targets reduces the number of additional fanout instructions that the compiler must insert. However, the delay to send and receive these targets may grow depending on the microarchitectural support for distributing operands. If target destinations are statically encoded, the ISA could restrict the location of potential consumers to enable efficient encoding of targets. This limits the flexibility with which instructions can be placed, however, which makes the placement problem more complex and may hurt performance in a distributed microarchitecture.

While a software fanout tree does increase the number of instructions, it also offers several advantages. First, the compiler can generate an unbalanced tree, providing critical instructions with their operands sooner than non-critical instructions. Second, with enough computational resources and careful instruction scheduling, a software fanout tree provides opportunities for parallelism in the lower parts of the tree that wide fanout instructions have difficulty providing without increased complexity in the microarchitecture. Finally, a software fanout tree provides maximum flexibility to the scheduler because it does not restrict the placement or ordering of the targets in any way. Here, we will explore the first benefit. We show how to apply Huffman and Hartley and Casavant's algorithms to build software fanout trees optimized for critical path length and compare them to optimizing solely for tree height. We show that minimizing the critical path length through a dataflow graph (Hartley and Casavant) yields better performance, but optimizing for total path weight (Huffman) accommodates inaccuracies in path length estimates while still giving priority to more critical instructions when compared to a balanced fanout tree.

4.1 Huffman software fanout tree construction

Our initial dataflow graph contains no fanout instructions. For each instruction that has more targets than can be encoded in the instruction, the scheduler inserts a software fanout tree consisting of MOV instructions, where each MOV instruction forwards its input to two output instructions. This algorithm generalizes to MOV instructions with higher fanout, but for simplicity we assume a MOV instruction can encode two targets. Figure 6(a) shows a dataflow graph before fanout insertion. This dataflow graph is extracted from a vector add kernel that distributes a base address to multiple load and store instructions, and to a loop increment. The portion of the dataflow graph below the target instructions is omitted for simplicity. In the graph, we annotate each instruction by the longest distance from it to a leaf instruction in the complete dataflow graph and show this weight in parenthesis in the figure.

If we ignore these weights and assign each leaf a weight of one, both algorithms produce the same fanout instruction dataflow graph which we call a balanced fanout tree, as shown in Figure 6(b). The critical path length for this graph is 23 cycles. This balanced fanout tree suffers because the targets of the read instruction have different costs. For fanout insertion, the cost is the longest distance from the target instruction to a leaf instruction in the dataflow graph, including both instruction execution latencies and known communication latencies. If the delays are exact, the best fanout tree minimizes this distance, and is produced by applying Hartley and Casavant.

For example, in Figure 6, the store instructions are close to the bottom of the graph and thus have low priority, while it is more critical to provide the fanout value to the load instructions because they are at the top of a chain with longer latency and also supply values to the store instructions. Since they are on the critical path, the code should execute faster if they receive their operands first.

To compute node weights, we first perform a post-order traversal of the dataflow graph without any fanout. A post-order traversal ensures that the children of an instruction have already been visited before the parent instruction is considered. Thus, the full distance to the bottom of the tree is known for all targets, including those that require fanout further down the dataflow graph.

When visiting an instruction that has more targets than it can encode, the fanout algorithm applies one of the Balanced, Huffman, or Hartley and Casavant algorithms. Figure 6(c) shows the fanout tree produced using Huffman's algorithm. The critical path length for the graph is now 22 cycles, and the total weight is 91 (the fanout instruction delay of one per fanout instruction plus the original weights).

Figure 6(d) shows the dataflow graph produced using Hartley and Casavant's algorithm. The critical path length is now 21 cycles, and the total weight is 91. In this case, the algorithms produce the same total weight (although Huffman will produce a lower total weight for many graphs), but

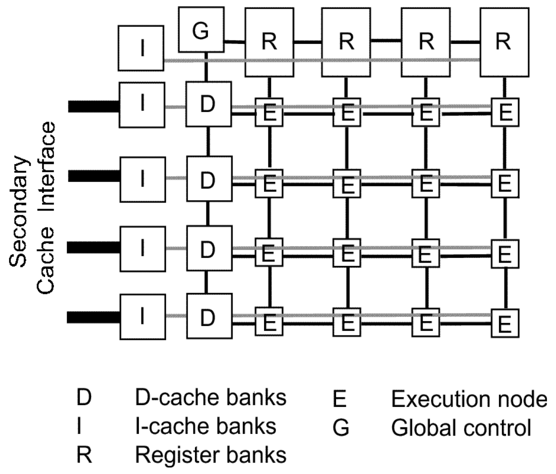


Figure 7: The TRIPS ALU grid

the critical path length is shorter with Hartley and Casavant. Two of the load instructions, however have a longer path length using this method than they do using the strict Huffman encoding. For a dataflow ISA where instructions execute as soon as their operands are ready and the dynamic critical path is unknown at compile time, it is unclear which method will produce the best performance. Minimizing the critical path is beneficial if critical path estimates are accurate, but in the presence of unknown delays that vary across different invocations of the dataflow graph, balancing the weighted sum can also offer benefits.

5. TRIPS–A Hybrid Dataflow Architecture

We evaluate these algorithms in the context of TRIPS, a prototype implementation of an EDGE (Explicit Data Graph Execution) [6, 7] ISA. EDGE architectures are designed to improve performance over superscalar designs as technology trends such as increased wire delay and power consumption become limiting factors. In this architecture, the atomic unit of execution is not an instruction, but a block of instructions which are mapped to a grid of ALUs at compile time. Within a block, TRIPS uses a dataflow ISA where instructions forward values directly from producer to consumer rather than communicating through a shared register file. Thus, the compiler encodes instruction dependences and placement, rather than forcing the microarchitecture to rediscover

Benchmark	Baer-Bovet tree height reduction	Huffman tree height reduction	Trees reduced	Average height before	Average height after
art_1	17.1	17.1	5	5	4
art_2	12.7	12.7	1	5	4
doppler_GMTI	4.9	4.9	2	4	3
fft4_GMTI	-1.5	-1.5	4	4	3
matrix_1	3.9	3.9	1	11	5

Table 1: Percent improvement in cycle count over no tree height reduction, number of affected trees, average height before and average height after reduction for TRIPS microbenchmarks.

dependences and decide where to execute instructions at runtime. Instructions can issue as soon as their operands arrive. This execution model simplifies the job of the compiler when compared to VLIW compilers, which have difficulty scheduling variable latency instructions.

TRIPS code benefits from tree height reduction because of the processor's ability to exploit ILP, and the direct target encoding of instructions. Tree height reduction can expose a great deal of parallelism (n/2 parallel instructions for a tree of n leaves) to feed the array of execution units. The TRIPS prototype executes blocks of 128 instructions atomically on an array of 16 ALUs as shown in Figure 7. This configuration allows most or all of the computations at each level of a tree to execute simultaneously. A potential problem with tree height reduction on conventional architectures is the increase in register pressure, since temporaries must be simultaneously stored. Since TRIPS uses direct instruction-to-instruction communication within a block, there is no additional register usage incurred by reducing the height of an expression tree.

6. Performance Evaluation

We measured the effect of Huffman-based expression tree height reduction and software fanout tree construction using a cycle-accurate simulator for TRIPS, which has been validated to within 5% of the RTL design [6]. Detailed simulation is extremely time-consuming (about 1000 instructions per second), so we evaluate performance using EEMBC (a benchmark suite for embedded systems) and a set of 24 microbenchmarks drawn from several sources. Fourteen of these programs are kernels from SPEC CPU2000 that account for at least 90% of the total execution time (ammp_1, ammp_2, art_1, art_2, art_3, bzip2_1, bzip2_2, bzip2_3, equake_1, gzip_1, gzip_2, parser_1, twolf_1, twolf_3).

The remaining benchmarks include five signal-processing kernels from a GMTI radar application (doppler_GMTI, fft2_GMTI, fft4_GMTI, forward_GMTI, transpose_GMTI), a 10×10 matrix-matrix multiply (matrix_1), an 8×8 discrete cosine transform (dct8x8), a 1024-element vector add (vadd) a prime number generator (sieve), and Dhrystone (dhry).

6.1 Arithmetic Expressions

We implemented arithmetic tree height reduction in Scale, a retargetable research compiler with a TRIPS backend [7]. Scale performs most classical optimizations on a machine-independent CFG representation. Tree height reduction occurs relatively late in the optimization sequence, to reduce the height of any expressions generated by other phases. In particular, unrolling loops containing accumulator variables creates opportunities for tree height reduction.

Of the 24 microbenchmarks, 19 have no arithmetic expressions that were candidates for reduction. Tree height reduction had no effect on the performance of these programs. Therefore, we focus our attention on the five programs with reducible trees: art_1, art_2, doppler_GMTI, fft4_GMTI and matrix_1. Table 1 shows the performance results for these programs, along with static statistics about the number of trees found and the average tree height reduction. For these programs, however, Huffman tree height reduction produces identical results to Baer and Bovet tree height reduction, suggesting that there may be few additional benefits gained from applying Huffman to arithmetic trees.

Tree height reduction significantly improves the performance of art_1 and art_2. These small programs have one main loop that computes the value of several floating-point accumulator variables over several hundred iterations. After unrolling each loop four times, the accumulator variable updates form narrow trees, which the compiler flattens to achieve high performance. Reduction is particularly effective in these cases for two reasons. First, the operations in the trees are relatively high-latency floating-point operations. Second, the loops' high iteration counts make reduction effective at reducing the critical path length of the entire program.

Of the remaining programs, doppler_GMTI and matrix_1 show improvements. These benchmarks also contain inner loops that compute accumulator variables, but the iteration count is lower

Benchmark	Baer-Bovet tree height reduction	Huffman tree height reduction	Trees reduced	Average height before	Average height after
Automotive					
a2time01	-0.1	0.4	0	0	0
aifftr01	0.0	-0.1	0	0	0
aifirf01	0.3	-0.2	0	0	0
aiifft01	0.0	0.0	0	0	0
basefp01	0.6	0.5	6	54	30
bitmnp01	2.6	2.6	0	0	0
cacheb01	-0.1	0.0	1	8	4
canrdr01	0.0	0.0	0	0	0
iirflt01	-1.0	-1.0	0	0	0
matrix01	-2.8	-2.9	3	24	12
pntrch01	0.0	0.0	0	0	0
puwmod01	0.2	0.0	0	0	0
rspeed01	-0.1	-0.1	0	0	0
tblook01	0.3	0.2	3	12	9
ttsprk01	0.1	0.0	1	4	3
Networking					
ospf	0.0	-0.1	3	12	9
pktflow	-0.3	-0.3	3	27	15
routelookup	-0.1	-0.1	0	0	0
Office					
bezier01	0.0	0.0	21	84	63
dither01	-3.3	-3.3	1	4	3
rotate01	0.0	0.0	0	0	0
text01	0.2	0.7	0	0	0
Telecom					
autcor00	0.1	0.3	1	9	5
conven00	0.1	0.0	0	0	0
fbital00	0.0	0.0	0	0	0
fft00	0.0	0.0	2	18	10
viterb00	0.0	0.0	0	0	0
Average	**-0.1**	**-0.1**	—	—	—

Table 2: Percent improvement in cycle count using Baer and Bovet or Huffman tree height reduction over no reduction, number of affected trees, average height before and average height after reduction for EEMBC benchmarks.

for each variable. The only negative result, fft4_GMTI, has small trees that compute local variables within a loop. Reducing these trees complicates scheduling and creates more network traffic at runtime, causing a net performance loss.

Table 2 shows the performance effect of tree height reduction on the EEMBC benchmarks. These effects are small due to the scarcity of arithmetic trees in critical portions of the programs.

Even bezier01, which has a comparatively large number of trees, shows no noticeable improvement due to tree height reduction. Some programs show small differences even though no trees are found, because our implementation sometimes stores a leaf value in a temporary register in preparation for balancing, and later discovers that the tree is too small to balance. Huffman and Baer-Bovet produce trees of equal height in these programs, differing only in the order of the leaves. These results seem to indicate the theoretical benefits of Huffman-based tree height reduction offer little performance improvement in real programs.

6.2 Software fanout tree insertion

The TRIPS compiler combines basic blocks into hyperblocks and produces an intermediate file where instructions are expressed in a RISC-like operand format [7]. The TRIPS instruction scheduler converts that operand format into target form, building a dataflow graph of the instructions within each hyperblock. The dataflow graph does not contain any fanout instructions, initially. Before mapping the dataflow graph onto the execution substrate, the scheduler is responsible for inserting a software fanout tree whenever an instruction has more targets than it can encode.

We implemented software fanout tree construction in the TRIPS scheduler using balanced fanout trees, the Huffman algorithm, and Hartley and Casavant's algorithm. To estimate the cost of each target instruction, we use the algorithm described in [17] to find the distance to the bottom of the tree adjusted for global effects. Table 3 provides results for each configuration on the EEMBC benchmark suite. We omit the cjpeg and djpeg benchmarks because their runtimes are intractably long, and we omit idctrn01 because it does not compile properly under our most aggressive compiler configuration.

Compared to balanced fanout trees, Huffman fanout trees provide an average 4.3% improvement. Fanout trees created using Hartley and Casavant's algorithm provide a 5.2% improvement when compared to balanced trees. On average, Hartley and Casavant fanout trees outperform Huffman fanout trees for this set of benchmarks. Because Hartley and Casavant's algorithm aggressively penalizes instructions that are not on the critical path in favor of those that are, unknown delays can sometimes create performance problems. For example, in the critical block of dither01 the scheduler

Benchmark	% improvement (cycles) Huffman	H/C	Average fanout tree height Balanced	Huffman	H/C	Max fanout tree height Balanced	Huffman	H/C
Automotive								
a2time01	1.4	1.8	3.5	3.7	3.7	7	7	7
aifftr01	5.8	5.0	3.4	3.6	3.6	6	9	9
aifirf01	1.1	2.0	3.6	3.9	3.9	7	8	8
aiifft01	6.2	6.4	3.4	3.5	3.5	6	7	7
basefp01	0.4	2.7	3.5	3.7	3.7	6	7	7
bitmnp01	9.8	8.1	3.7	3.8	3.8	7	8	8
cacheb01	-0.9	0.8	3.7	3.9	3.9	6	7	7
canrdr01	9.2	9.5	3.5	3.7	3.7	7	8	8
iirflt01	-3.3	-2.3	3.6	4.0	4.0	6	8	8
matrix01	5.3	5.6	3.5	3.7	3.7	6	8	8
pntrch01	9.5	6.7	3.6	3.9	3.9	7	7	7
puwmod01	9.6	10.1	3.5	3.7	3.7	6	7	7
rspeed01	7.2	8.5	3.5	3.7	3.7	6	7	7
tblook01	4.6	4.7	3.5	3.8	3.8	6	9	9
ttsprk01	8.4	8.9	3.4	3.7	3.7	7	7	7
Networking								
ospf	8.6	9.5	3.6	3.9	4.0	6	9	12
pktflow	6.4	7.0	3.6	3.7	3.9	6	9	12
routelookup	8.5	9.7	3.7	4.0	4.2	7	8	8
Office								
bezier01	8.3	8.5	3.2	3.2	3.4	5	6	15
dither01	0.4	-4.8	3.2	3.3	3.4	6	7	8
rotate01	6.7	8.9	3.7	3.8	4.0	6	6	7
text01	2.9	3.2	3.4	3.6	3.7	6	8	10
Telecom								
autcor00	0.7	1.4	3.6	3.8	3.9	6	7	11
conven00	-1.9	-0.4	3.6	3.8	3.9	6	7	9
fbital00	-3.2	1.8	3.3	3.4	3.6	6	7	9
fft00	5.2	17.9	3.4	3.7	3.9	6	8	11
viterb00	-0.0	0.6	3.5	3.7	3.8	7	9	12
common	-	-	3.6	3.9	3.9	6	9	9
Average	**4.3**	**5.2**	**3.5**	**3.7**	**3.8**	**6.2**	**7.6**	**8.8**

Table 3: Percent improvement in cycle count over balanced trees, total number of trees, average number of targets, and ratio of fanout instructions to total instructions for EEMBC benchmarks.

underestimates the importance of a loop-carried dependence, making its weights inaccurate. Hartley and Casavant's algorithm hurts performance by building an aggressively unbalanced tree that favors the wrong target. The Huffman fanout algorithm, in contrast, creates a more balanced tree that alleviates this penalty. When the scheduler estimates the relative priority of the critical path correctly, as in fft00, the Hartley and Casavant algorithm performs extremely well, and the Huffman

Fanout tree statistics for EEMBC benchmarks				
Benchmark	Num trees	Avg num targets	Fanout ratio	Max fanout
Automotive				
a2time01	258	0.2	4.1	18
aifftr01	634	0.3	3.4	30
aifirf01	818	0.3	4.1	26
aiifft01	458	0.3	3.3	30
basefp01	114	0.2	4.3	16
bitmnp01	668	0.3	3.8	17
cacheb01	151	0.2	4.2	16
canrdr01	776	0.3	4.0	26
iirflt01	686	0.3	4.2	16
matrix01	2024	0.3	3.9	16
pntrch01	341	0.3	3.8	20
puwmod01	442	0.3	3.6	16
rspeed01	177	0.2	3.8	16
tblook01	411	0.3	3.5	15
ttsprk01	446	0.2	3.4	28
Networking				
ospf	500	0.3	4.3	24
pktflow	581	0.3	3.6	13
routelookup	419	0.4	3.9	18
Office				
bezier01	79	0.2	3.1	19
dither01	108	0.2	2.9	15
rotate01	292	0.3	3.9	14
text01	614	0.3	3.3	14
Telecom				
autcor00	121	0.3	3.6	14
conven00	305	0.3	4.1	11
fbital00	144	0.2	3.2	18
fft00	444	0.3	4.2	17
viterb00	275	0.3	3.9	17
common	2340	0.3	3.7	17
Average	**522.3**	**0.3**	**3.8**	**18.4**

Table 4: Static fanout statistics including the number of instructions that require fanout, the average number of targets of an instruction requiring fanout, the ratio of fanout instructions to total instructions after fanout insertion, and the maximum instruction fanout.

algorithm does not realize the same benefit because it is not as aggressive in optimizing the critical path.

To evaluate the differences among the trees generated by each algorithm we gathered static data about the fanout trees, as well as aggregate data across the entire benchmark suite. We use static

Fanout distribution			
Number of targets	Occurrences	Number of targets	Occurrences
1	547	16	32
2	4888	17	10
3	3058	18	12
4	2511	19	5
5	983	20	1
6	916	21	0
7	406	22	0
8	511	23	0
9	225	24	2
10	215	25	0
11	70	26	7
12	97	27	0
13	56	28	1
14	37	29	2
15	32	30	2

Table 5: Distribution of number of targets for all instructions that require fanout aggregated across all EEMBC benchmarks.

Benchmark	**Fanout tree height distribution**		
Tree height	Balanced	Hartley and Casavant	Huffman
0	0	0	0
1	0	0	0
2	0	0	0
3	8711	7987	7962
4	3773	3214	3130
5	1835	2124	2092
6	291	950	999
7	16	295	316
8	0	44	72
9	0	12	27
10	0	0	14
11	0	0	3
12	0	0	10
13	0	0	0
14	0	0	0
15	0	0	1

Table 6: Distribution of tree height for all fanout trees aggregated across all EEMBC benchmarks.

statistics, and factor out the EEMBC benchmarks share a set of common files that are executed infrequently by most benchmarks. We separate these common files and show their statistics in the rows labeled common. In Table 3, Columns 4-6 show the average fanout tree height, and Columns

7-9 show the maximum fanout tree height for each configuration. The fanout tree height is the maximum number of inserted MOV instructions along any path from the root of a fanout tree to a leaf (two, four, six, and six in Figures 6(a), 6(b), 6(c), and 6(d), respectively). As expected, Hartley and Casavant's fanout algorithm produces the most unbalanced trees in order to create a globally minimized critical path. Benchmarks that show a particularly large performance improvement, such as fft00 and ospf, also demonstrate significant variations in maximum and average tree height.

In Table 4, Column 2 shows the number of instructions in each benchmark that require fanout to forward their values, and Column 3 shows the fraction of the instructions in the final program that are fanout instructions. We can mitigate these relatively high fanout ratios by using fanout instructions with more targets, for example three or four, but these instructions impose scheduling constraints that we leave for future work. Column 4 shows the average number of leaves in an instruction fanout tree. Finally, Column 5 shows the maximum fanout for any instruction.

Table 5 shows the aggregate distribution across all the benchmarks of the number of leaves in fanout trees. We see values that have only one target yet require fanout because read instructions cannot directly target write instructions. Thus, one fanout instruction must be inserted between the read and the write instruction even if the read does not have any other targets. Many instructions with two targets require fanout because some instructions, including those that encode an immediate operand, can only target one instruction. This table shows the distribution of fanout across EEMBC.

Table 6 shows the distribution of the fanout tree height aggregated across the entire EEMBC suite. We only count instructions where fanout is necessary, thus the minimum is a height of three (if the height is two then no fanout was inserted). This table shows the frequency with which each algorithm generates trees of each height. The maximum fanout of 30 in Table 5 corresponds to a maximum balanced tree height of seven rather than six in Table 6 because the root of the tree can only target one instruction, increasing the height by one. The balanced tree algorithm clearly produces the shortest trees, while Huffman creates slightly unbalanced trees. Hartley and Casavant's algorithm, in contrast, sometimes produces extremely unbalanced trees in an attempt to minimize the global critical path length.

7. Conclusions

We have described a variant of Huffman encoding that minimizes the critical path length of an expression consisting of associative and commutative operators. This optimization creates opportunities for constant folding and optimally balances trees even in the presence of intermediate values that must be preserved. We cast software fanout tree construction in a dataflow architecture as a tree height reduction problem where the cost of the leaves is non-uniform. We apply the Huffman algorithm to this problem and contrast it with a similar algorithm that minimizes tree height with variable cost leaves. We consider the effects of unknown execution latencies on software fanout tree construction, observing that the Huffman algorithm tolerates unknown latencies, but does not always aggressively minimize the critical path length.

Variable cost leaves places a new restriction on the tree height reduction problem that may also be applicable to arithmetic expression trees. Arithmetic tree height reduction in software typically assumes unit cost for the input values. In reality, the arrival time of the inputs to the arithmetic expression tree will vary, and an unbalanced tree may actually allow the maximum ILP. With good cost estimates in the compiler, the tree height reduction problem could be extended to account for these variable costs in programs with sufficient opportunity for tree height reduction.

Our results indicate that there are few opportunities for arithmetic tree height reduction in the EEMBC benchmarks and SPEC2000 kernels, and that previous algorithms successfully exploit those opportunities. Careful fanout insertion, on the other hand, improves average performance by 5% for the TRIPS architecture. The tree height reduction algorithm we present here is applicable to VLIW and superscalar architectures that also need to exploit ILP to perform well. Our analysis of software fanout tree construction techniques applies to any EDGE or dataflow architecture that uses software fanout trees, and is also applicable to any tree height reduction problem where arrival times can be estimated, but are not known statically.

References

[1] J.-L. Baer and D. P. Bovet, "Compilations of arithmetic expressions for parallel computations," in *IEIP Congress*, (Amsterdam, Holland), pp. 340–346, Nov. 1968.

[2] J. C. Beatty, "An axiomatic approach to code optimization for experssions," *Journal of the ACM*, vol. 19, Oct. 1972.

[3] D. J. Kuck, ed., *The Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.

[4] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing*, (Minneapolis, MN), pp. 808–817, 1992.

[5] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 242–251, June 1989.

[6] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and others, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, pp. 44–55, July 2004.

[7] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley, "Compiling for EDGE architectures," in *International Symposium on Code Generation and Optimization*, (Manhattan, NY), pp. 185–195, Mar. 2006.

[8] B. Landwehr and P. Marwedel, "A new optimization technique for improving resource exploitation and critical path minization," in *Symposium on System Synthesis*, (Antwerp, Belgium), pp. 65–72, Sept. 1997.

[9] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *Journal of the ACM*, vol. 17, pp. 715–728, Oct. 1970.

[10] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098–1101, Sept. 1952.

[11] K. Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[12] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the Second International Symposium on Computer Architecture*, (New York, NY, USA), pp. 126–132, 1975.

[13] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Proceedings of the 36th Symposium on Microarchitecture*, December 2003.

[14] R. Hartley and A. Casavant, "Tree–height minimization in pipelined architectures," *IEEE Transactions on Computer Aided Design*, vol. 8, pp. 112–115, 1989.

[15] J. S. Vitter, "Design and analysis of dynamic Huffman codes," *Journal of the ACM*, vol. 34, pp. 825–845, Oct. 1987.

[16] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[17] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. McKinley, "A spatial path scheduling algorithm for edge architectures," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.