Dataflow: A Complement to Superscalar

Mihai Budiu
mbudiu@microsoft.com
Microsoft Research, Silicon Valley

Pedro V. Artigas
artigas@cs.cmu.edu

Seth Copen Goldstein
seth@cs.cmu.edu

Carnegie Mellon University

Abstract

There has been a resurgence of interest in dataflow architectures, because of their potential for exploiting parallelism with low overhead. In this paper we analyze the performance of a class of static dataflow machines on integer media and control-intensive programs and we explain why a dataflow machine, even with unlimited resources, does not always outperform a superscalar processor on general-purpose codes, under the assumption that both machines take the same time to execute basic operations. We compare a program-specific dataflow machine with unlimited parallelism to a superscalar processor running the same program. While the dataflow machines provide very good performance on most data-parallel programs, we show that the dataflow machine cannot always take advantage of the available parallelism. Using the dynamic critical path we investigate the mechanisms used by superscalar processors to provide a performance advantage and their impact on a dataflow model.

1. Introduction

The renewed interest in dataflow architectures is in part sparked by the underlying elegance of the model, but also motivated by the changes wrought by continued technology scaling. Increased silicon resources [1], the desire to avoid global wires [16], the complexity of global clock distribution [3], the diminishing returns of ever longer pipelines [37], and the complexity of the many monolithic superscalar structures [25] all suggest designs should be simpler, rely mostly on local signals, and contain many instances of each type of functional unit. How effectively these "distributed" architectures can exploit instruction-level parallelism (ILP) is still an open question, that we are addressing in this paper.

This paper addresses this question by analyzing an architecture that is at an extreme in the space of possible solutions. We call this architecture ASH, an acronym for Application-Specific Hardware. In ASH, functional units are never shared between instructions. ASH is optimized for communication: since computation units are never shared, there is no need to share communication channels, and each data value uses a dedicated one-directional high-speed link. We use the CASH compiler [6] to translate automatically each C program into a hardware static dataflow machine, synthesized as collection of asynchronous circuits. These circuits feature completely distributed datapath, register file, and control logic, with no global signals, and without any notion of a clock. Programs run on ASH use very little power, with energy efficiencies up to four orders of magnitude better than superscalar processors [6], but their performance is sometimes worse than when they are run on a superscalar. To understand the performance problems of the dataflow machines, we have developed a new tool which allows us to visualize and analyze the dynamic critical path of programs executed on ASH. Our analysis shows that ASH circuits have certain bottlenecks inherent in the dataflow model, bottlenecks that a superscalar overcomes using specific mechanisms, that traditionally were not considered as part of the dataflow model.

For completeness we start in Section 2 by describing the compilation methodology. In Section 3 we employ timing-accurate simulation to measure the performance of C programs on both superscalar and dataflow machines, observing that dataflow's exploitation of unlimited parallelism does not always offer a performance advantage. In our comparison we employ a simple normalizing assumption: that all arithmetic operations take the same time in both machines. In Section 4 we describe a fully automatic tool to visualize the dynamic critical path of a dataflow execution. This tool is used in Section 5 to identify performance bottlenecks by analyzing several representative hot functions. We show how various architectural features impact the performance of the dataflow machines and how they compare to superscalar microarchitectural mechanisms. The ability of modern superscalars to speculate branches, to speculate loads, to perform dynamic register renaming and dynamic loop unrolling, to issue the same instruction multiple times, and to execute procedure calls before their arguments have been computed are instrumental for high performance. Implementing some of these mechanisms in a dataflow machine requires extensions to the basic model. In fact, some of the mechanisms require monolithic structures and global wires, attributes contrary to the distributed nature of dataflow that is exploited by the recent proposals.

2. Compiling C to Hardware

This section describes how CASH translates C programs into hardware dataflow machines. CASH represents the input program using Pegasus [4, 5], a dataflow intermediate representation (IR). The output of CASH is a hardware dataflow machine which directly executes the input program [45, 6].

A program is represented by a directed graph in which nodes are operations and edges indicate value flow. CASH can generate drawings of the program representation using the `dot` language [15], which we used to generate all the figures in this paper. We briefly describe how programs represented as control-

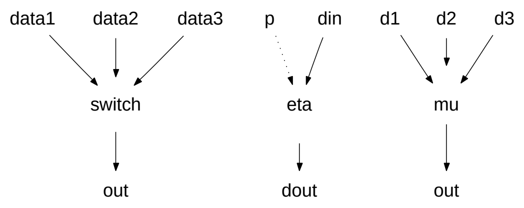


Figure 1: *Control-flow operations:* MU, SWITCH, ETA. *Dotted lines represent Boolean values.*

flow graphs are translated to Pegasus. The translation is accomplished in two steps: first, the control-flow graph is divided into acyclic regions. The current implementation uses hyperblocks [23] as acyclic regions, trying to grow hyperblocks as large as possible. Each region is separately converted to a predicated data flow representation. The second phase of the translation stitches hyperblocks together, creating the representation for a whole procedure.

From Hyperblocks to Dataflow. Each hyperblock is transformed into straight-line code through the use of predication, using techniques similar to PSSA [8]. Within hyperblocks Pegasus uses multiplexor (MUX) nodes instead of SSA ϕ-nodes.

Speculation is introduced by predicate promotion [22]: instructions without side-effects are executed unconditionally once a hyperblock is entered. Predication and speculation are thus core constructs in Pegasus. The former is used for translating control-flow constructs into dataflow; the latter for reducing the criticality of control-dependences [20]. They effectively increase ILP.

Some operations have *lenient* implementations: they can produce an output before all inputs are known. E.g., a Boolean AND can generate a *false* result as soon as any input is known to be *false* [45].

Steering Dataflow: MU, SWITCH and ETA. Pegasus uses three special computational primitives to steer data between hyperblocks. Figure 1 shows their graphic representation. MU operations have n data inputs and one data output. At any moment at most one of the n data inputs is available. That input is copied to the output immediately. Pegasus uses MUs at the entry points of a hyperblock. MU nodes can have a second output; the *index* of the input that last received data, whose only purpose is to drive SWITCH nodes.

SWITCH nodes serve the same purpose as MU nodes, having n data inputs and one data output. However, they also have a *control input*, which indicates which of the data inputs is expected to fire next. SWITCHes are similar to MUXes, in the sense that the control input selects which of the data inputs to forward. Unlike MUXes, they expect only one data input to be present at a time.

CASH generates a special value called the *current execution point*, abbreviated *crt*, which corresponds somewhat to a traditional "program counter", and indicates the currently executing hyperblock. The invariant maintained is that there exists exactly a single instance of a *crt* value in the whole program at any one point. The *crt* value is always steered by MU nodes; the *crt* MU nodes provide the control for the SWITCH node inputs.

While MU and SWITCH nodes are used to steer data at the entry of a hyperblock, ETA nodes are used to steer data out of a hyperblock. Each ETA has two inputs — one for data and one for a predicate — and one output. An ETA forwards the data to the output if the predicate is *true*; if the predicate is *false*, the data is consumed and nothing is forwarded.

Memory Accesses are represented through explicit LOAD and STORE nodes. These operations, as others with side-effects, also have a predicate input: if the predicate is *false*, the operation is not executed.

The compiler adds dependence edges, called *token edges*, to explicitly synchronize operations whose side-effects may not commute. Operations with *memory side-effects* (LOAD, STORE, CALL, and RETURN) all have a token input. Token edges explicitly encode data flow through memory. An operation with memory side-effects must collect tokens from all its potentially conflicting predecessors (e.g., a STORE following a set of LOADs). The COMBINE operator is used for this purpose. COMBINE has multiple token inputs and a single token output; it generates an output after it receives all its inputs.

For this paper we assume that all memory operations are connected to a single load-store queue (LSQ), which interfaces the dataflow machine to a conventional memory hierarchy. We implement an aggressive protocol for memory access: once a memory operation has received a token, it reserves the next slot in the LSQ. When additional inputs reach the operation (e.g., address, predicate, or data for STOREs), it updates its reserved slot. The LSQ performs dynamic disambiguation even with incomplete information. As soon as the LSQ slot reservation succeeds, the operation issues its token output, therefore enabling its successors to reserve slots themselves. Given enough bandwidth to the LSQ and enough storage space in the LSQ itself, the token part of the computation will propagate ahead of the scalar computation. Note that scheme is very similar to the operation of LSQ in a superscalar processor [34].

Synthesis. The hardware back-end synthesizes each node as a distinct micropipeline stage [38], having a single output register, and each edge as a *channel* connecting the source and destination pipe stages [45]. Each channel is composed of a data bus, a data-ready signal, and an acknowledgment signal, used by the consumer(s) to indicate they are ready to receive another value.

At run-time, each edge of the graph either holds a value or is empty. Once all required inputs of an operation are available, it can begin computing. The computation consumes the input values and when the output is empty, it can latch the newly produced output. This behavior corresponds to a static dataflow machine: i.e., each edge can hold at most one value in the single register.

Example. Figure 2 shows a function that uses `i` as an induction variable and `sum` to accumulate the sum of the squares of `i`. On the right is the program's Pegasus representation, which consists of three hyperblocks. Hyperblock 1 receives the *crt* and `n` as arguments and initializes `i` and `sum` to 0. Based on the comparison node labeled with **(A)**, ETA nodes steer values to either hyperblock 2 or 3. Hyperblock 2 represents the loop; it contains three SWITCH nodes, one for each of the loop-carried values, `i`, `n` and `sum`, and a MU node for *crt* (labeled **(B)** in the figure), which drives the SWITCH nodes. Back-edges within a hyperblock denote loop-carried values; in this example there are four such edges in hyperblock 2. Hyperblock 3 is the function epilog, containing two MERGE nodes — one for the return value and one for *crt*— and the RETURN.

```
int squares(int n)
{
  int i = 0,
      sum = 0;

  for (i=0;i<n;i++)
      sum += i*i;
  return sum;
}
```

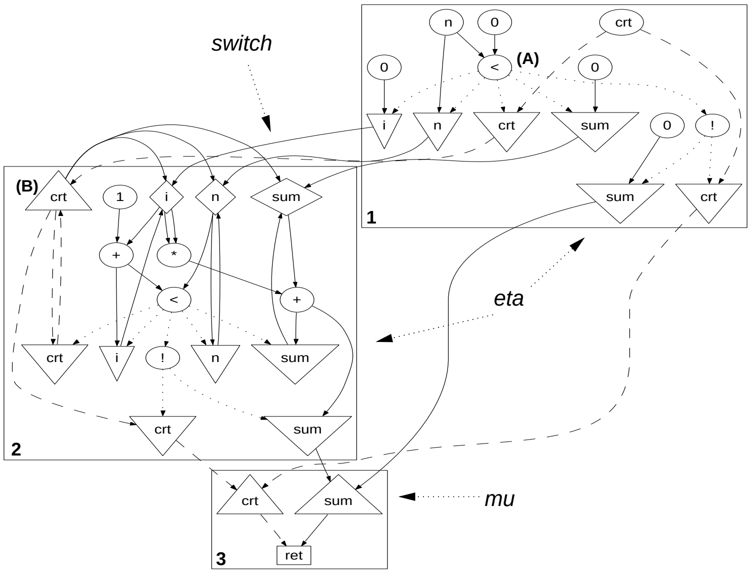


Figure 2: *A C program and its Pegasus representation; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicates. The dashed lines are used to represent the flow of* crt.

3. Performance Comparison

We compare ASH and superscalar by examining whole programs executions on timing-accurate simulators. Since there are many parameters, one should interpret this comparison as a limit study.

Experimental Setup. We make the following assumptions: (1) arithmetic operations have the same latencies on both computational fabrics; (2) MUXes, SWITCHes, MUs and Boolean operations have latencies proportional to the log of the number of inputs; (3) ETA takes the same time as an addition; (4) memory operations in ASH, as opposed to the superscalar, incur an additional cost for network arbitration and propagation to the load-store queue (LSQ); (5) the ASH latency through the memory access network is 1 cycle, independent on the program size; (6) the memory hierarchy used for both models is identical: a load-store queue (LSQ) and a two-level cache hierarchy. The L1I$ latency for the CPU is 1 cycle; the L1D$ latency is 2 cycles for both models. For this study we use a similar LSQ for both fabrics. The LSQ has two input and two output ports. We model the queueing for the LSQ ports. The L2$ latency is 8 cycles, and the memory latency is 72 cycles.

Some of these assumptions are clearly optimistic for ASH when dealing with whole programs. In particular, the complexity of the memory access network and of the interconnection medium for procedure calls and returns cannot be assumed constant. In the best case the latency of these networks should grow as $\sqrt{n}$, where n is the size of the compiled program, assuming a two-dimensional layout.

The superscalar is a 4-way out-of-order SimpleScalar simulation [7], using the off-the-shelf simulator, with a 4-wide issue, 5-stage pipeline, with the PISA instruction set; we use `gcc` 2.7.2.3 with `-O2` optimization flags, generating MIPS binaries. The dataflow machine is simulated by using a high-level simulator which is automatically generated by CASH. CASH is configured to compile with maximum optimizations, and performs extensive loop unrolling.

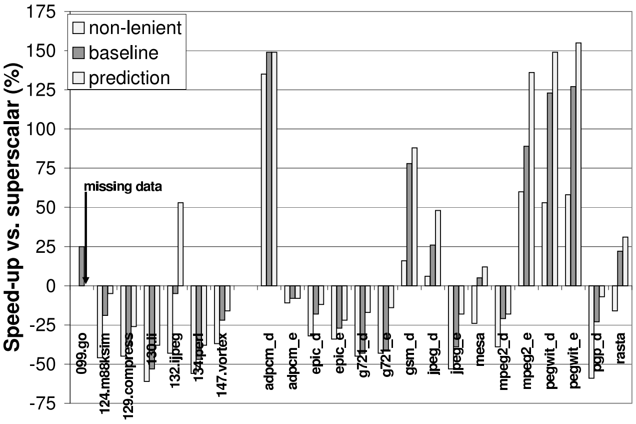


Figure 3: *Speed-up (percents) of Mediabench and SpecInt95 benchmarks executed on various dataflow machine configurations. The reference is a superscalar four-way out-of-order processor. Negative values indicate a slower execution on dataflow. The first bar shows non-lenient execution performance (c.f. Section 5.3). The second bar correspond to the baseline data-flow machine performance. The last bar shows the effect of perfect* ETA *predicate prediction (c.f. Section 5.1, Section 5.4).*

Performance Measurements. We present data for all the benchmarks that successfully compiled and simulated on our infrastructure. Naively we would expect that ASH is always faster—since it benefits from (a) unlimited parallelism, (b) lack of computational resource constraints, (c) dynamic scheduling, and (d) no overhead for instruction fetch/decode/dispatch. However, only some programs show performance improvements (Figure 3). The rest of this paper is devoted to explaining these results.

4. The Dynamic Critical Path

To understand the above results we characterize the program execution by its *dynamic critical path*. This path is simultaneously a function of program dependences, runtime execution path (which is a function of the input data), hardware resources and dynamic scheduling. Despite the apparent simplicity of the concept, only recently has a practical methodology been proposed to extract and analyze dynamic critical paths in superscalar processors [14, 13].

The key insight is that all critical events must be *last arrival events*. Such an event is the last one which enables data to be latched. Events correspond to signal transitions on the edges of the dataflow graphs. Most often, the last-arrival event is the last input to reach an operation. However, for lenient operations (see Section 2) the last arrival event is the input that enables the computation of the output. If a circuit experiences backlog, the last arrival event may be the *acknowledgment* signal which enables the computation to proceed. (The acknowledgment signal is the complement of the stall signal in the superscalar pipeline.) In a typical execution, multiple critical events may correspond to the same hardware structure (i.e., the input to an operation may be critical several times). One way to summarize the critical path is as a *histogram of edges*, where each edge is labeled with the number of occurrences.

Since datapath hardware is never shared in ASH, the critical path maps very naturally to program operations. This has enabled us to completely automate the computation of the critical path. We have implemented a utility that captures and post-processes the complete execution trace produced by the simulator. First, the tools filters out all but the last arrival events for each operation. Then, the last arrival events trace is reversed and a continuous chain of last arrival events is computed starting with the last operation executed. This chain is the dynamic critical path. Finally, the tool generates a drawing of a specified function with the critical path edges highlighted proportionally to their frequency of occurrence. Throughout the rest of this paper we rely on this tool to understand performance differences between the superscalar processor and ASH.

5. Insights from the Critical Path

In this section we use the critical path to investigate the impact on performance of various microarchitectural and compiler mechanisms. We discuss several hot functions which exhibit poor performance on ASH, and whose behavior is representative for other program fragments.

5.1. Outer Loop Pipelining

Branch prediction, in addition to reducing pipeline bubbles, is instrumental in creating loop parallelism.

Figure 4 shows a code fragment from the `epic_d` Mediabench program whose performance on ASH is about 66% of the superscalar due to the lack of branch prediction. There are three branches involved in two very shallow nested loops: (1) the `do` backwards branch, mostly not taken (this loop iterates either once or twice); (2) the `if` branch, almost always taken; and (3) the `for` backwards branch, almost always taken. There are 8 machine instructions in the outer loop body, and the processor window is large enough to hold several iterations in flight. The innermost loop contains a tight long-latency true loop-carried data dependence, between `swap_pos` and its new value, which apparently cannot be accelerated unless some form of value prediction is used.

```
internal_int_transpose(int *mat,int rows,int cols){
  register int modulus = rows*cols - 1;
  register int swap_pos, current_pos, swap_val;

  for(current_pos=1;current_pos<modulus;current_pos++)
   {
      swap_pos = current_pos;
      do {
          swap_pos = (swap_pos * cols) % modulus;
      } while (swap_pos < current_pos);

      if ( current_pos != swap_pos ) {
          swap_val = mat[swap_pos];
          mat[swap_pos] = mat[current_pos];
          mat[current_pos] = swap_val;
      }
   }
}
```

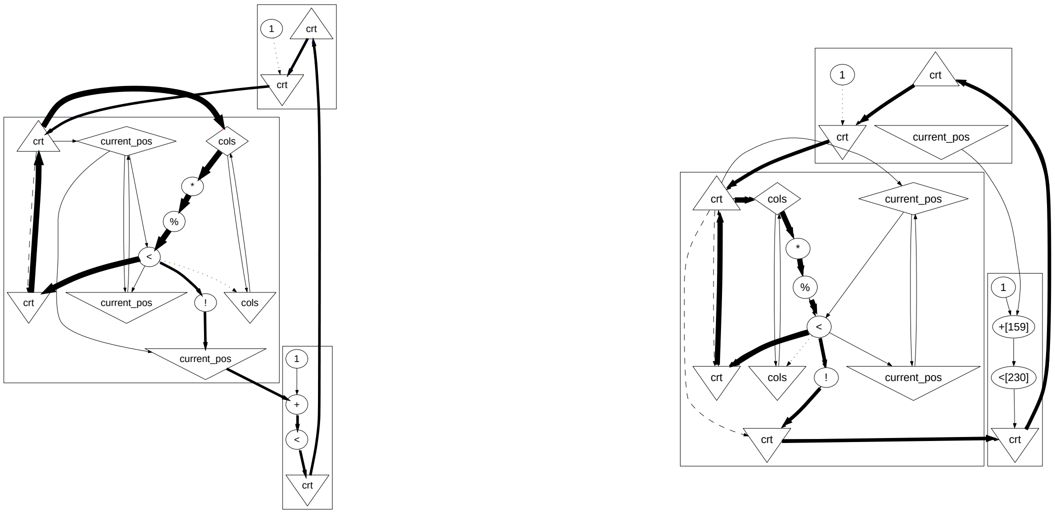
Figure 4: *The* `internal_int_transpose` *function from the* `epic_d` *benchmark from Mediabench, which exhibits poor performance on ASH.*

However, by using branch prediction the processor can effectively issue a second iteration of the *outermost* `for` loop before completing the previous one. The iterations of the outer `for` are effectively pipelined. The critical resource in this loop is thus the single division unit, which is not pipelined, and takes 24 cycles to complete a modulus. Indeed, the throughput of this function averages 27 cycles per outer loop iteration on the CPU. The cost of most of the other operations in the loop (e.g., multiply, branches, memory accesses) is hidden by the division.

In ASH the most important part of the critical path is shown in bold lines in Figure 5(a). (We have omitted all nodes that are not on the critical path.) The thickest edges have a higher frequency. The performance of this code is hampered by the explicit sequencing of control along the three hyperblocks. In particular, a new iteration of the outer loop cannot be initiated as long as the inner loop has not terminated. The predicates controlling both the loops (the < operations) are on the critical path. This also puts the three-cycle multiplier on the critical path. The extra operations on this path sum up to 12 cycles, which corresponds to the observed 33% slowdown. The inner loop "<" is the "last" value computed within the innermost loop and it is immediately used to decide whether to loop again. The outer loop comparison "<" could be computed much earlier, since both its inputs are known. However, the value of `current_pos`, which is unchanged within the `do` loop, is not *propagated* out of the inner loop until its iterations have completed.

5.2. Bypassing Across Control-Equivalent Hyperblocks

The hyperblocks surrounding the inner loop in Figure 5(a) are control-equivalent, so the computation of the condition of the outer loop can be lifted to be performed in parallel with the inner loop. This optimization is called *bypass* [17], since the invariant value



(a) The original critical path alternates between the innermost cycle and the outer loop. The MU node for *crt* in the bottom hyperblock has been removed by the optimizer, since it has a single input.

(b) Critical path after bypassing. The computation of the outer loop predicate is no longer on the critical path, since it is performed in parallel with the inner loop.

Figure 5: *Dynamic critical path of the function* `internal_int_transpose`.

```
/* x >  0  - 66% */
/* x <= 0  - 33% */
if (x > 0)
    y = -x;
else
    y = b*x;
```

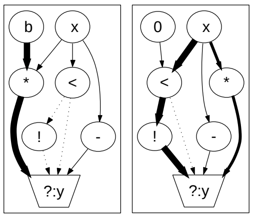


Figure 6: *Sample program fragment and the corresponding critical path with strict and lenient execution. When strict execution is employed, the multiplier is on the critical path; with a lenient implementation of the* MUX*, the multiplier is critical only when its result is used by latter computations.*

of `current_pos` is forwarded directly without crossing the inner loop. Bypassing allows the static data flow machine to exploit control-independence, a graph property that cannot be expressed in regular ISAs [20], but can aid in improving program performance [30, 10]. This observation was one of the original motivations of Multiscalar Processors [36], which employ multiple threads for this purpose.

Performing this optimization changes the critical path as shown in Figure 5(b). Nevertheless the performance of the code fragment improves by only 4.5%, since we have removed only two inexpensive operations. The critical path along the outer loop is now solely composed of the machinery required to control the global flow of the computation.

5.3. Lenient Execution

The form of speculative execution employed by Pegasus, which executes all forward branches simultaneously, alleviates the impact of branches, but may be plagued by the problem of *unbalanced paths*, well-known from the literature on predicated execution [2], as illustrated in Figure 6: if a MUX waits for all inputs to generate the result, the critical path of the entire construct always includes the longest of the critical paths of the inputs. We use leniency to solve this problem. MUXes are implemented leniently: as soon as a selector is *true* and the corresponding data is available, a MUX generates its output. A result of leniency is that *the dynamic critical path is the same as in a non-speculative implementation*. For example, when the multiplication in Figure 6 is not used, it does not affect the critical path.

Figure 3's first bar illustrates ASH's lackluster performance in the absence of lenient execution.

5.4. Control Dependences and Predicates

Branch prediction not only decreases pipeline bubbles and increases parallelism, it also can eliminate control dependences, as Figure 7 shows. In this example ASH is two times slower than the processor. Figure 8(a) shows the essential part of the optimized code generated by CASH with the critical path highlighted. As in the case of the previous function, the predicate deciding whether the loop should iterate is again the last thing computed, because it depends on the loaded value. The critical path contains not only the LOAD but the LOAD predicate computation as well, since it indicates whether the LOAD needs to be executed or not.

Figure 8(b) shows the critical path when the value of the `&&` (the ETA predicate) is perfectly predicted. Under the assumption that the LOAD is not executed speculatively, the critical path now goes through the LOAD predicate, which prevents the LOAD from being issued early to memory.

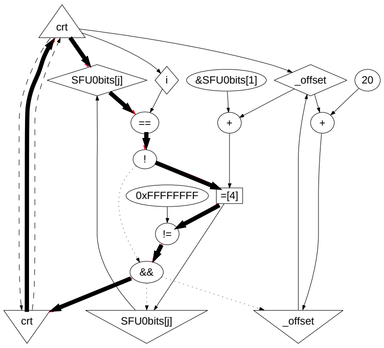
```
void init_processor(void)
{
  int i, j;
  for(i = 0; i < 64; i++)
  {
    for(j = 0; SFU0bits[j].regnum != 0xFFFFFFFF; j++)
       if(SFU0bits[j].regnum == i) break;
    m88000.SFU0_regs[i] = SFU0bits[j].reset;
  }
}
```

(a) Source code.

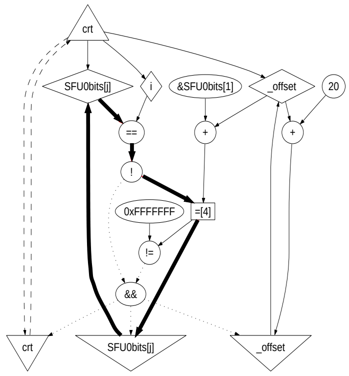
```
LOOP:
L1:   beq $v0,$a1,EXIT
L2:   addiu $v1,$v1,20
L3:   lw $v0,0($v1)
L4:   addiu $a0,$a0,1
L5:   bne $v0,$a3,LOOP
EXIT:
```

(b) `gcc` MIPS assembly for the inner loop.

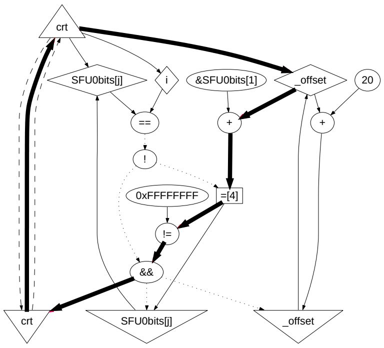
Figure 7: *The function* `init_processor` *from the SpecInt95 benchmark* `124.m88ksim`*.*



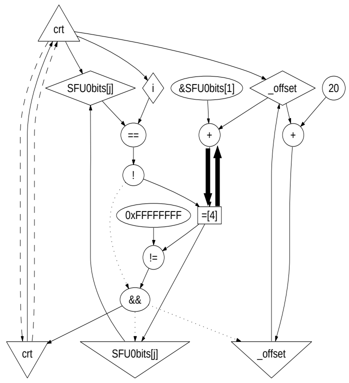
(a) Original critical path



(b) Path with perfect ETA predicate prediction.



(c) Critical path with speculative loads.



(d) Path with both predicate prediction and speculative LOADs. The most frequent edges form a tight loop: LOAD address $\rightarrow$ LOAD $\rightarrow$ ack edge to LOAD address.

Figure 8: *Dynamic critical path of the function* `init_processor`*. The* LOAD *fetches a 4-bytes word, and is shown as* `=[4]`*.*

5.5. Issuing Loads Speculatively

We have modified the LSQ connected to the dataflow machine, allowing it to initiate a LOAD that reaches the head of the queue speculatively, i.e., without waiting for a confirmation that the actual predicate value is *true*. After this change the critical path contains the address computation, as shown in Figure 8(c). The problem is that the address computation for the next iteration is not made speculatively — it needs to know that the next iteration is initiated before starting the computation.

The superscalar code is shown in Figure 7(b). The code contains a loop-carried dependence involving four of the five instructions (L1$\stackrel{control}{\longrightarrow}$L2$\stackrel{true}{\longrightarrow}$L3$\stackrel{true}{\longrightarrow}$L5$\stackrel{control}{\longrightarrow}$L1), so the latency through this loop ought to be at least four cycles. However, if branch prediction correctly guesses the outcome of the two branches, the control dependences vanish from the critical path, and the only remaining circular dependence is between L2 and itself. A 5-wide processor with good branch prediction can execute this loop in only one cycle per iteration!

Going back to ASH, when combining LOAD speculation and ETA predicate prediction, the critical path is shown in Figure 8(d). The critical path now contains an acknowledgment edge, from the LOAD to its address computation. Technically the limiting dependence is an anti-dependence between the output register of the LOAD and itself, from two consecutive iterations. This dependence stems from the fact that the LOAD has a single output register, and therefore cannot acknowledge its address input before it has successfully latched its result. However, despite hitting in the cache, the latency of the LOAD is more than two cycles — it takes one cycle just to get to the LSQ, and then two more cycles to hit into L1, and one more cycle to return the result. Each loop iteration latency is *bounded below* by the latency of the longest operation in the loop, which in this case is the LOAD.

A superscalar processor removes the load anti-dependence by register renaming: each issued load writes its result to a different physical register. It can therefore have multiple instances of the same load operation in flight. If the memory system supports pipelined load operations, it can sustain a very high throughput for this loop. This proves that a dynamic dataflow model (i.e., which allows multiple values for each graph edge), which is well approximated by a Superscalar when enough computational resources are present, is strictly more powerful than a static model — the one employed in our dataflow machines.

We can address this in the static dataflow model in two ways:

(1) Pipelining the LOAD operation itself to allow multiple simultaneous accesses initiated by the same operation. This solution requires either hardware in the LOAD that must be able to buffer and handle out-of-order replies from the memory system for multiple outstanding LOAD requests (i.e. a reorder buffer), or a simpler FIFO buffer and a memory system that returns replies in-order.

(2) Using loop *unrolling*, which transforms one LOAD into multiple LOADs made in parallel. The latency of the computation is still bounded below by the latency of a LOAD, but the limit now applies to code containing multiple iterations of the original loop, so the throughput is increased. This option is discussed next.

5.6. Dynamic Loop Unrolling

When we statically unroll the `init_processor` loop and use perfect branch prediction, the performance on ASH approaches but still does not equal the superscalar processor. The reason is that the inner loop executes different numbers of iterations at different times: sometimes it iterates many times, but sometimes it iterates only once. Static unrolling only helps when the number of iterations is large. This is exacerbated by the fact that in CASH the unrolling of inner loops creates a large loop whose body is a single hyperblock—all of whose instructions must be executed, due to predicate promotion and speculative execution. Thus, when the loop has a single iteration, all the extra code from unrolling is pure overhead.

The superscalar does well on this loop without compiler unrolling, because it performs *dynamic unrolling*: it unrolls the loop at run-time inside the instruction window, as guided by the branch predictor. The only overhead it pays is when prediction fails.

```
int unused_arg(int unused, int used)
{ return used; }

int main()
{
    ....
    return unused_arg(EXPENSIVE(i), i);
}
```

Figure 9: *Sample synthetic code exhibiting reduced performance due to the strictness of the* CALL *and* RETURN *instructions.*

5.7. Call Strictness

The program in Figure 9 shows how two superscalar mechanisms provide an advantage in the implementation of procedure calls: (1) the decoupling of control transfer and argument forwarding, and (2) out-of-order execution. In this example the function `unused_arg` ignores its first argument, which is computed using some expensive expression. In the current version of ASH, the CALL instruction is a single circuit element, and has a *strict* implementation, i.e., it waits for all arguments to be available before transferring control to the callee. Therefore, the critical path contains the expensive computation.

The superscalar passes the call arguments in registers and the call instruction is just a branch. The net effect is that the call instruction is *lenient*, since control may be passed to a procedure before all its arguments are computed. In this example, the procedure may even return before the expensive computation is terminated, because its result is unused.

A similar problem plagues the current ASH implementation of the RETURN instruction: this is strict in all three of its arguments — the value returned, the memory token, and *crt*. In contrast, the processor just writes the return value to a register and branches back to the caller. The return branch may complete much earlier than the actual register write-back. If the caller ignores the returned value (a frequent occurrence in C functions), the returned value computation may vanish from the critical path (assuming it does not stall the processor through the use of resources or through the in-order commit stage.)

The simplest solution to this problem in the dataflow machine, not always applicable, is to inline the callee. We are also considering CALL/RETURN implementations which allow lenient invocations.

5.8. Synchronization Overhead

Another feature of ASH which has a detrimental impact on performance is the run-time cost of steering the data flow. This machinery, which is not required by the superscalar, adds significant overhead to the execution cost. First, ETAs, which roughly correspond to conditional branches in a superscalar, incur a propagation delay. This contrasts with the superscalar which relies on branch prediction to significantly reduce the cost of conditional branching. Moreover superscalar microarchitectures may rely on branch folding [12] to execute many branches with truly zero cost. Figure 10's first bar illustrates the performance gains assuming ETAs incur no delay; performance improvements are noticeable.

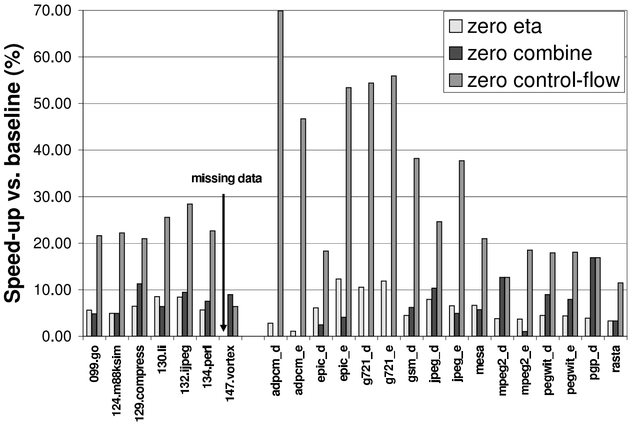


Figure 10: *Speed-up (percents) of Mediabench and SpecInt95 benchmarks executed on various dataflow machine configurations. The reference is the baseline data flow machine. The first and second bar assume, respectively, that* ETA *and* COMBINE *have no cost. The last bar shows the impact of eliminating the cost of control-flow operations.*

In ASH control-flow graph join points also require to extra machinery, namely SWITCH, MU, and MUX nodes. These nodes correspond roughly to *labels* in assembly language and have a cost logarithmic in the number of inputs, whereas in the superscalar labels are essentially free. Figure 10's last bar corresponds to ASH performance under the assumption these operations incur zero delay. As the performance improvements demonstrate, there is a significant amount of extra cost associated with control flow join points in ASH. Note that some proposed processor architectures supporting predication may also inject "multiplexor" micro-operations [47], which will incur a run-time cost. (This is also true of the "conditional move" instructions.)

Even the COMBINE operator, which is used to enforce run-time memory dependences, sometimes contributes to the critical path. The cost of a COMBINE is logarithmic in the number of inputs. COMBINE operators with a large fan-in, which can arise because of an ambiguous reference or because of a STORE following a large set of LOADs, can incur a substantial run-time overhead. Figure 10's second bar correspond to the achievable performance assuming COMBINEs have zero cost.

We believe this is a fundamental issue: the tension between parallelism and synchronization overhead, that is manifested here at the level of individual instructions. Dataflow replaces global information (e.g., the program counter or the branch predictor) with explicit communication and synchronization. The steering of data by ETA, MUX, MU, and SWITCH nodes requires coordination between multiple nodes, and also a broadcast of the branch condition. Even though the cost of each of these operations is low, it still becomes an important overhead in very tight loops, such as the ones exemplified in this section. Other distributed architectures, such as MIT's Raw, also have to pay for broadcasting the branch conditions to the remote computations [21]. This is one of the reasons that executing control-intensive code on a single Raw tile often is faster than parallelizing the code across multiple tiles [40].

6. Related Work

One can interpret these results as a limit study on the amount of ILP available in C programs. Many studies exploring the interplay between architectural features, compilers and ILP have been carried during the course of the years [20, 46, 41, 27, 36, 9, 35, 42, 24, 29, 19, 26, 18, 43, 39]. We study the behavior of large C programs using timing-accurate simulation of both a superscalar processor and a static dataflow machine; memory bandwidth is also modeled. We now discuss the distinguishing aspects of this work.

Memory Disambiguation / Dependence Analysis: Memory dependences are a known stumbling block to exposing ILP. Trace based studies found a significant increase in the available parallelism when relying on perfect memory disambiguation [41, 46]. Our study relies only on realistic, compiler based, memory dependence analysis coupled with run-time disambiguation in the load-store queue.

Control Dependences: Exploiting control dependences allows a larger amount of ILP to be exposed without relying on speculation. Trace based studies quantified the impact of exploiting control dependences to increase the available parallelism and found significant gains [20, 41]. A less ambitious approach is to enable early execution of instructions from control equivalent basic blocks. Rotenberg [30] studied the effectiveness of this approach in the context of a superscalar processor, in this work we also exploit this technique in the context of a data flow machine; more recently Chen-Yong [10] proposed a superscalar microarchitecture, Skipper, that also exploits control-independence. Instead of speculating on hard-to-predict branches Skipper executes control-equivalent instructions; which are always executed regardless of the branch outcome. Execution of instructions control dependent on the branch proceed after the hard-to-predict branch is resolved.

Branch prediction: Branch predition coupled with speculative execution is another sucessfull technique to reduce the impact of control flow on paralelism [48, 49]; we evaluate the impact of no branch prediction and perfect branch prediction on dataflow.

Instruction Window Size and ILP: Our static data flow machine exploits all the parallelism that can be identified by our compiler; constrasting with other studies in which parallelism is extracted dynamically from a limited size parametrizable window [46, 41].

Dataflow Computation: Recent publications that compare WaveScalar [39] — a model of computation similar to our dataflow architecture — with a very wide superscalar have reached quite different conclusions from ours, implying that a dataflow model of execution has the potential for high parallelism on integer-type benchmarks. WaveScalar assumes a model of execution closer to dynamic dataflow, but uses a significantly different memory access protocol to ensure coherence. As this work shows, performance differences cannot be easily ascribed to a single aspect of the architecture; a detailed evaluation would need to be carried out to compare the two models in a fair way.

The TRIPS project [31] studies a hybrid form of execution, somewhere between dynamic dataflow and superscalar processors: there is a very wide instruction word which drives a dataflow limited-size fabric. This work attempts to accurately models the cost of communication across the 2-D lattice of functional units. Exploiting ILP in control-intensive programs is quite difficult for this type of architecture as well. The numbers in [32] show sustained IPC below 2 for SpecInt95, even with perfect memory dependence prediction; a monolithic memory disambiguation engine [11] can obtain only around 80% of the benefits. Implementing a distributed memory disambiguation mechanism may also incur some additional synchronization costs, corresponding to our COMBINE operators.

There is also a large amount of work on dataflow computation [44], but the vast majority used parallel functional languages, and thus cannot be used for a fair comparison with our implementation.

7. Conclusions

From the performance results and from our analysis of some critical code segments, we conclude that in our model of dataflow execution there are some fundamental limitations towards the exploitation of ILP:

(1) Control dependences still limit ILP, as they do for non-predicated code [20]. It is well known that predication, which we employ aggressively, does not completely eliminate unpredictable branches, but only propagates unpredictability to other branches [33].

(2) Implementing a generic prediction scheme (be it branch prediction or value prediction) in a dataflow model is hindered by the difficulty of building a mechanism for squashing the computation on the wrong paths. The register renaming mechanism of a superscalar provides such a mechanism for free. An interesting proposal for a distributed dataflow implementation of speculation was made in [28].

(3) Even assuming control-flow speculation can be squashed, good quality prediction often requires "global" information (e.g., two-level branch prediction, memory dependence prediction, etc.). Such an implementation may not naturally map to the distributed nature of ASH dataflow machines.

(4) The ability of register renaming to remove anti-dependences between the same operation in different iterations is crucial for efficiently executing tight loops.

(5) The distributed nature of the computation in a dataflow machine requires more remote accesses (i.e., even LSQ accesses become non-local), which are more expensive than in a monolithic system.

(6) The cost of the explicit "join" operations in the representation (MU, SWITCH, MUX, COMBINE), all of which are essentially "free" in a processor, can be substantial. This is the cost of *synchronization* which replaces the global signals in monolithic architectures.

This study has increased our admiration for the capability of superscalar processors of exploiting ILP and pipeline parallelism, through the virtualization of a very limited set of resources: computational units, registers, and bandwidth on both internal and external interconnection networks. Since there is no free lunch, the price paid for the effectiveness in exploiting ILP is power consumption: 90% of the die of Pentium 4 (even excluding the caches) is devoted only to *support* structures, trying to keep the functional units busy. The dataflow model we have presented lies at the other extreme: it lacks *all* helper structures, can provide excellent performance on code with ample data parallelism, but may be as much as two times slower than a superscalar for control-intensive code. However, as we show in [6], ASH uses 1000 times less power to perform the same tasks. We believe that there is a place for both models of computation.

We conclude that for control-intensive code, with relatively low ILP, the superscalar performance is excellent compared to a dataflow machine. Performance-wise, dataflow is a winning solution for programs exhibiting a large amount of parallelism, which it can execute with high performance and low power. More research is needed for designing a balanced hybrid which can inherit more of the strengths of both models of computation.

References

[1] International technology roadmap for semiconductors (ITRS). http://public.itrs.net/Files/1999_SIA_Roadmap/Design.pdf, 1999.

[2] D. I. August, W. m. W. Hwu, et al. A framework for balancing control flow and predication. In *International Symposium on Computer Architecture (ISCA)*, December 1997.

[3] D. W. Bailey and B. J. Benschneider. Clocking design and analysis for a 600-MHz Alpha microprocessor. *IEEE Journal of Solid-State Circuits*, 33(11):1627, November 1998.

[4] M. Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[5] M. Budiu and S. C. Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, September 2002.

[6] M. Budiu, G. Venkataramani, et al. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[7] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25, pages 13–25. ACM SIGARCH, June 1997.

[8] L. Carter, B. Simon, et al. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[9] D.-K. Chen, H.-H. Su, et al. The impact of synchronization and granularity in parallel systems. In *International Symposium on Computer Architecture (ISCA)*, pages 239–248, 1990.

[10] C.-Y. Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. In *MICRO*, pages 4–15. IEEE Computer Society Press, December 2001.

[11] R. Desikan, S. Sethumadhavan, et al. Scalable selective re-execution for EDGE architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 120–132, 2004.

[12] D. R. Ditzel and H. R. McLellan. Branch folding in the crips microprocessor: Reducing branch delay to zero. In A. Press, editor, *International Symposium on Computer Architecture (ISCA)*, pages 2–9. ACM Press, 1987.

[13] B. Fields, R. Bodík, et al. Slack: Maximizing performance under technological constraints. In *International Symposium on Computer Architecture (ISCA)*, pages 47–58, 2002.

[14] B. A. Fields, S. Rubin, et al. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture (ISCA)*, 2001.

[15] E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software Practice And Experience*, 1(5), 1999. http://www.research.att.com/sw/tools/graphviz.

[16] R. Ho, K. Mai, et al. The future of wires. *the IEEE*, 89(4):490–504, April 2001.

[17] R. Johnson and K. Pingali. Dependence-based program analysis. In *PLDI*, pages 78 – 89, June 1993.

[18] A. Klauser, A. Paithankar, et al. Selective eager execution on the PolyPath architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 250–259, June 1998.

[19] D. Kuck, Y. Muraoka, et al. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers (TOC)*, C-21:1293–1310, 1972.

[20] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *International Symposium on Computer Architecture (ISCA)*, 1992.

[21] W. Lee, R. Barua, et al. Space-time scheduling of instruction-level parallelism on a Raw machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 46–57, 1998.

[22] S. A. Mahlke, R. E. Hauk, et al. A comparison of full and partial predicated execution support for ILP processors. In *International Symposium on Computer Architecture (ISCA)*, pages 138–149. ACM, May 1995.

[23] S. A. Mahlke, D. C. Lin, et al. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Computer Architecture (ISCA)*, pages 45–54, Dec 1992.

[24] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers (TOC)*, C-33(11):968–976, November 1984.

[25] S. Palacharla, N. P. Jouppi, et al. Complexity-effective superscalar processors. In *International Symposium on Computer Architecture (ISCA)*, pages 206–218, June 1997.

[26] M. A. Postiff, D. A. Green, et al. Limits of instruction level parallelism in SPEC95 applications. In *Workshop Interaction between Compilers and Computer Architectures*, October 1998.

[27] L. Rauchwerger, P. K. Dubey, et al. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–117. IEEE Computer Society Press, 1993.

[28] R.Desikan, S.Sethumadhavan, et al. Lightweight distributed selective re-execution and its implications for value speculation. In *First Value Prediction Workshop*, June 2003.

[29] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers (TOC)*, C-21(12):1405–1411, December 1972.

[30] E. Rotenberg, Q. Jacobson, et al. A study of control independence in superscalar processors. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[31] K. Sankaralingam, R. Nagarajan, et al. A technology-scalable architecture for fast clocks and high ILP. In *Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[32] K. Sankaralingam, R. Nagarajan, et al. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 422–433. ACM Press, 2003.

[33] B. Simon, B. Calder, et al. Incorporating predicate information into branch predictors. In *First Workshop on EPIC Architectures and Compiler Technology*, December 2001.

[34] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *the IEEE*, 83(12):1609–1624, 1995.

[35] M. D. Smith, M. Johnson, et al. Limits on multiple instruction issue. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 290–302. ACM Press, 1989.

[36] G. S. Sohi, S. E. Breach, et al. Multiscalar processors. In *International Symposium on Computer Architecture (ISCA)*, volume 23 (2). ACM, May 1995.

[37] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[38] I. Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6):720–738, June 1989.

[39] S. Swanson, K. Michelson, et al. WaveScalar. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2003.

[40] M. B. Taylor, W. Lee, et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *International Symposium on Computer Architecture (ISCA)*, June 2004.

[41] K. B. Theobald, G. R. Gao, et al. The effects of resource limitations on program parallelism. Advanced Compilers, Architectures and Parallel Systems (ACAPS) Technical Memo 52, January 1993.

[42] G. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers (TOC)*, C-19:885–895, October 1970.

[43] A. K. Uht, D. Morano, et al. Levo — a scalable processor with high IPC. *Journal of Instruction-Level Parallelism*, 5, August 2003.

[44] A. H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.

[45] G. Venkataramani, M. Budiu, et al. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic Syntheiss*, June 2004.

[46] D. W. Wall. Limits of instruction-level parallelism. Technical Report Digital WRL Research Report 93/6, Digital Western Research Laboratory, November 1993.

[47] P. H. Wang, H. Wang, et al. Register renaming and scheduling for dynamic execution of predicated code. In *International Symposium on High-Performance Computer Architecture (HPCA)*, January 2001.

[48] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 129–139, December 1992.

[49] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *International Symposium on Computer Architecture (ISCA)*, pages 257–266, 1993.