Appears in the Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture

Dataflow Predication

Aaron Smith Ramadass Nagarajan Karthikeyan Sankaralingam Robert McDonald
Doug Burger Stephen W. Keckler Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin
{asmith, ramdas, karu, robertmc, dburger, skeckler, mckinley}@cs.utexas.edu

Abstract

Predication facilitates high-bandwidth fetch and large static scheduling regions, but has typically been too complex to implement comprehensively in out-of-order microarchitectures. This paper describes dataflow predication*, which provides per-instruction predication in a dataflow ISA, low predication computation overheads similar to VLIW ISAs, and low complexity out-of-order issue. A two-bit field in each instruction specifies whether an instruction is predicated, in which case, an arriving predicate token determines whether an instruction should execute. Dataflow predication incorporates three features that reduce predication overheads. First,* dataflow predicate computation *permits computation of compound predicates with virtually no overhead instructions. Second,* early mispredication termination *squashes in-flight instructions with false predicates at any time, eliminating the overhead of falsely predicated paths. Finally,* implicit predication *mitigates the fanout overhead of dataflow predicates by reducing the number of explicitly predicated instructions, by predicating only the heads of dependence chains. Dataflow predication also exposes new compiler optimizations–such as disjoint instruction merging and path-sensitive predicate removal–for increased performance of predicated code in an out-of-order design.*

1. Introduction

Predication linearizes instruction flows by converting control dependences to data dependences, thus improving control flow predictability, instruction fetch bandwidth, and the size of the instruction scheduling window for the compiler. VLIW and vector machines have successfully applied predication to obtain all three of these improvements [12, 23, 28].

Despite its advantages, predicated execution has not achieved widespread use in out-of-order architectures. The complexities of merging predication with dynamic scheduling [21]–particularly register renaming [27]–have outweighed its perceived benefits. Consequently, many ISAs supporting dynamically scheduled implementations–such as Alpha and SPARC–provide only limited support for predication using conditional moves and stores.

Due to scaling limits of conventional superscalar designs, researchers have recently begun investigating architectures that combine dataflow-like behavior with conventional programming models [7, 8, 25]. Historical dataflow machines typically implemented only limited *partial predication*, using *gate* and *switch* operators [2, 13, 26]. Subsequent advances in predication, used primarily in VLIW architectures, present an opportunity to reduce predication overheads for these new dataflow-like machines.

This paper describes *dataflow predication*, which provides lightweight ISA support for predicating any instruction in a dataflow-like architecture. In dataflow predication, any instruction producing a value can instead produce a predicate. A two-bit field in each instruction specifies whether that instruction must receive a matching predicate token to issue. With this ISA support, as well as appropriate support in the microarchitecture and compiler, dataflow-like ISAs can exploit the benefits afforded by both predication and dynamic out-of-order issue, at much lower complexity than superscalar architectures. Dataflow predication incorporates three features that enable low predication overhead:

- *Dataflow predicate computation*: Since any instruction can receive a predicate, the compiler need not gate the data operands for an instruction explicitly–unlike prior dataflow architectures–but may instead simply predicate the consumer. By eliminating gate and switch operators, per-instruction dataflow predication reduces the dependence path height, and permits low-overhead predicate inversions (NOT), conjunctions (AND), and disjunctions (OR), with bipolarized predicates, pred-

icated test instructions, and the routing of multiple predicates to a single instruction, respectively.

- *Implicit predication*: Although the ISA supports predication of any instruction, the compiler need not predicate all instructions within a predicated basic block explicitly. The compiler may predicate only the head instruction in a dependence chain, thus implicitly predicating its successors.

- *Early mispredication termination*: Dataflow predication supports squashing of instructions on a false predicate path at any time. This capability prevents the dependence height of falsely predicated paths from reducing performance, and is also necessary to support implicit predication.

These three features of dataflow predication lend themselves to several compiler optimizations that reduce predicate overheads and improve performance:

- *Predicate fanout removal*: The major source of overhead for dataflow predication is fanning out predicates to potentially many consumers. In dataflow-like architectures, this communication requires building a software fanout tree to distribute the predicate to its consumers. To reduce this overhead, the compiler may apply either speculative hoisting or implicit predication, predicating only the tails or heads of dependence chains, respectively. Both techniques reduce the number of instructions to which predicates must be communicated.

- *Path-sensitive predicate removal*: Using inter-block liveness analysis, the compiler can remove the predicate from an instruction whose result is unused on the path that is complementary to the predicate.

- *Disjoint instruction merging*: The compiler can merge identical instructions on disjoint control flow paths, eliminating redundant instructions and exposing opportunities for further optimization.

The TRIPS architecture, which implements dataflow predication, is one instantiation of an EDGE (Explicit Data Graph Execution) architecture [8]. An EDGE architecture has two distinguishing features: (1) it supports *block-atomic* execution, in which statically defined blocks of instructions must commit atomically–either all of a block's instructions commit, or none of them do. (2) Within each block, the EDGE ISA implements dataflow communication between instructions, in which dependences are explicitly encoded by specifying target locations. The ISA instantiates all predicates as dataflow arcs, and does not rely on a centralized predicate register file.

In addition to the ISA and microarchitectural support necessary for an implementation of dataflow predication in the TRIPS prototype, this paper describes the compiler algorithms and optimizations necessary to support efficient predication. The TRIPS compiler generates predicated code by performing if-conversion on basic blocks, inserting predicates, and merging the multiple basic blocks into hyperblocks. As the compiler forms hyperblocks, it applies scalar predicate optimizations to produce more compact and efficient hyperblocks. Simulation results show that these dataflow predicate optimizations improve performance by 12% over an aggressively predicated baseline.

Dataflow predication makes possible a clean synergy between predication and out-of-order execution, with lower predicate overhead than the partial predication implemented in previous dataflow architectures, and lower hardware complexity than proposed predicated out-of-order superscalar designs.

2. Prior Predicate Optimizations

Architectures have used predication since the 1970s. The CRAY-1 implemented predication in the form of vector masks [23] to guard individual vector operations. Predicated execution became more prevalent in VLIW machines in the 1980s and 1990s. The Multiflow Trace machines supported partial predication using the select instruction [17]. The Cydra 5 [22] and the IA-64 Intel Itanium processors' ISAs include a predicate operand with every instruction. Several RISC architectures also support some predicated execution; the in-order ARM processor predicates most instructions, but the out-of-order Alpha and SPARC V9 architectures limit predication to conditional move instructions.

Predication research has generally fallen into two categories: ISA and microarchitecture support for efficient execution, and compiler algorithms and optimizations to use and exploit predication. Allen et al. first described if-conversion to convert control dependences to data dependences [1]. Mahlke et al. proposed the use of hyperblocks as an effective compiler structure for performing predication and exposing scheduling regions to the compiler [20]. Researchers have also shown that predication is effective for enabling software pipelining on loops with control structures [12, 28].

Several solutions have been proposed to alleviate the overheads of predication in VLIW architectures and to a limited extent, in dynamic superscalar architectures.

Fetch and execution overhead: Previous processors must issue a predicated instruction even if its execution is nullified by the guarding predicate, resulting in wasted fetch and possibly execution bandwidth that can otherwise be utilized by useful instructions. In addition, an instruction that is dependent on the predicate value cannot execute until that

predicate is computed. In VLIW machines, instructions that do not execute consume and waste issue slots, potentially elongating the schedule. August et al. propose a framework that mitigates this problem by balancing control speculation and predication [5]. To alleviate these problems in out-of-order processors, researchers have proposed *predicate prediction*, which predicts the resolution of the predicate in the dispatch logic [9], *wish branches*, which enable the hardware to dynamically and selectively employ predicated execution [16], and *predicate slip*, which delays the use of the guard predicate until commit [27].

Register renaming: Predication complicates the task of dynamic register renaming in an out-of-order processor due to multiple potential definitions along if-converted control paths. The definition, or the lack thereof, cannot be detected until its guarding predicate is resolved. Researchers have proposed several solutions for this problem, including predicate prediction [9] and using operators such as conditional select [17], μ-op [15], and select-μop [27]. Even with these solutions, renaming remains prohibitively complex.

Predicate registers: Conventional architectures typically save the results of predicate-defining instructions in either the general purpose register space or in a private predicate namespace. In addition to specifying two (or more) data source operands, an predicated instruction must also specify its predicate operand. In IA-64, the predicate operand consumes six bits of each instruction. Due to this encoding pressure, some architectures use predication only in a small number of instructions (ARM is a notable exception). For example, Alpha and SPARC V9 architectures offer a single conditional move operation for use in simple control constructs. To extend predication to other instructions, Pnevmatikatos et al. propose using the GUARD instruction [21]. This instruction, in conjunction with a predicate register file, specifies which instruction to guard among the set of successor instructions.

Predicate computation: To generate predicates for instructions inside complex control structures, the compiler must invert and merge predicates generated along each if-converted branch. A long predicate computation chain, in addition to increasing instruction overhead, may end up on the program critical path. Researchers have addressed this problem in different ways: by generating complementary predicates [14], by using *wired operators* [14], and by *program decision logic minimization* [4].

Of the conventional architectures, VLIW architectures benefit from low-overhead predication, but lose performance because falsely predicated instructions can lengthen the critical path of execution. Superscalar processors have not benefited from predication due to the complexity of its implementation in an out-of-order microarchitecture. Less conventional architectures, such as historical dataflow machines, have combined only partial predication with dynamic scheduling.

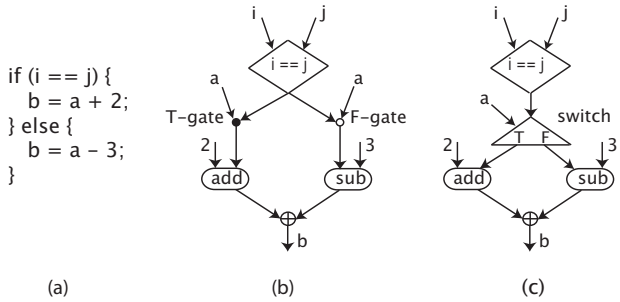

Figure 1. Conversion of (a) an if-then-else construct using (b) a T-gate and F-gate, and (c) a switch instruction.

2.1. Dataflow Predication

The conversion of control dependences to data dependences is essential for dataflow execution. Static dataflow machines used *T-gate* and *F-gate* operators [13]. The T-gate (F-gate) operator copies an input to its output if and only if its control input operand carries a true (false) value, otherwise it does not produce any output. The MIT-tagged token dataflow machine combined the functions of these two operators using the *switch* operator, which conditionally steers an input operand to either of two destinations based on a control input [2, 26]. The destination instruction that does not receive its input does not execute.

Figure 1 illustrates how the compiler converts an if-then-else construct to dataflow using the gate and switch operators. If `i` is equal to `j`, the T-gate passes the value of `a` to the target `add` instruction, as shown in Figure 1b. Conversely, if `i` does not equal `j`, the F-gate absorbs `a`, produces no output, and consequently the `sub` instruction does not execute. Figure 1c shows how the compiler transforms the same code using the switch instruction. If `i` equals `j`, the switch instruction steers its data input `a` to the `add` instruction, instead of the `sub` instruction. Thus only the `add` instruction executes. The join operator depicted by $\oplus$ in both Figure 1b and Figure 1c is a logical placeholder in the dataflow graph, indicating a single producer for the variable `b`; it does not represent an instruction. By inhibiting the delivery of input operands, the compiler guards the execution of instructions within if-then-else statements.

The switch and gate operators, when applied naively, can limit the parallelism in dataflow machines. For example, gating every input within a conditional statement results in significant overhead. In addition, the gate operators serialize the execution of the succeeding instructions. For example, in Figure 1, the `add` and `sub` instructions can execute only after the preceding gate and switch operators, limiting

parallelism. Prior research in dataflow has not eliminated these overheads. While Beck et al. propose techniques to eliminate unnecessary switch instructions [6], researchers have not applied the full range of optimizations developed for VLIW [5] to dataflow architectures.

Dataflow predication, as instantiated in the TRIPS architecture, differs from previous partially predicated dataflow architectures in three major ways: First, predicates may directly guard individual instructions, avoiding the need for gate or switch instructions. Second, any instruction can generate a predicate merely by targeting the predicate operand of another instruction. Third, instructions may receive multiple predicate operands before firing. These three features enable dense encoding, as each 32-bit instruction requires only two bits to specify whether it is predicated. They also enable efficient compound predicate computation, since dataflow predication supports the disjunction of an arbitrary number of predicates, and since predicate-producing instructions may themselves be predicated. Finally, they support implicit predication, since only the input instructions to a dataflow graph need to be predicated to implicitly predicate the entire graph.

3. TRIPS ISA Support for Predication

An EDGE architecture has two distinguishing features. First, it supports *block-atomic* execution, in which statically defined blocks of instructions must commit atomically; either all instructions of a block complete and commit, or none of them do. Second, within each block, it supports dataflow execution with *direct-instruction communication*, in which the ISA explicitly encodes the dependences.

In the TRIPS EDGE prototype, a block completes when it produces a consistent set of outputs; for each execution, a block must write to the same registers, execute the same number of stores, and generate one branch target. The TRIPS hardware counts these outputs and signals block completion when all have been produced. Instructions communicate between blocks through registers and memory.

Within a block, instructions use direct instruction communication, which includes all predicates. Each instruction contains the identifiers of instructions that depend on its result, not the operands of the instruction itself. The dataflow execution model dictates that an instruction executes only when all of its operands are sent to it by its parent instructions. When an instruction completes execution, it sends its result directly to other instruction *targets* within the same block. Figure 2 shows an example of a simple C-code fragment of an if-then-else and a diagram representing its if-converted dataflow graph. The right-most field in the instruction is the target identifier, which specifies the consumer of the instruction's result. The TRIPS ISA allows up to 128 instructions within each block.

Any instruction that produces a value (arithmetic operations, comparisons, etc.) must specify at least one target, each of which uses a 9-bit target encoding. Seven bits of the target identify one of 128 possible instruction targets within the block, while the remaining two bits indicate either that the result is either the left, right, or predicate operand of the target instruction.

3.1. Predication Rules

The TRIPS ISA but must follow a number of rules to produce well-formed, predicated blocks:

1. Any instruction (except for a few specific data movement and constant generation instructions) may be predicated. A two-bit predicate field indicates whether an instruction is predicated and on what polarity of the arriving predicate the instruction should be executed.

2. For a predicated instruction to fire and execute, it must receive all of its data operands and a matching predicate operand. A matching predicate is one that matches the polarity of the waiting instruction. For example, an instruction waiting for a "false" predicate will only fire when a "false" predicate arrives.

3. Multiple instructions may target the predicate operand of an instruction, but at most one may deliver a matching predicate. This capability permits aggressive instruction merging.

4. To deliver a predicate to multiple predicated instructions, the compiler must construct the necessary fanout tree using a series of multi-target `move` instructions.

5. The predicated dataflow graphs must preserve the exception behavior of an unpredicated program, meaning that the same exceptions must be detected at the TRIPS block boundaries.

3.2. Predicate Encoding

Dataflow predication provides the ability to compute compound predicates efficiently, while also exploiting early mispredication detection and implicit predication. The overheads of dataflow predication, as instantiated in TRIPS, require a two-bit field per instruction and fanout instructions for routing predicates to more than two predicate consumers. Previous architectures also had significant overheads for predication. Partially predicated dataflow ISAs added extra split and merge instructions. VLIW architectures required larger per-instruction fields (e.g. six bits) to specify predicate registers.

Figure 2 illustrates predicate generation. The test instruction (`teq`) receives `i` and `j`, computes a predicate, and

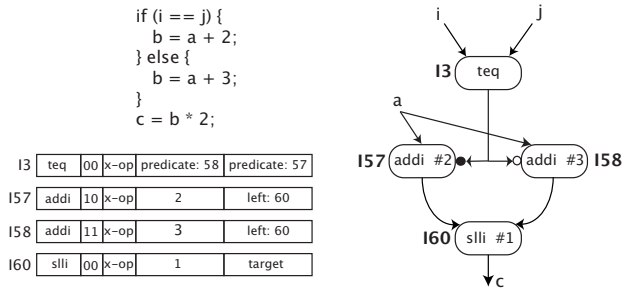

Figure 2. Predication in the TRIPS ISA.

sends it to the two `addi` instructions. Note that the `addi` instructions are predicated on opposite polarities; the black circle indicates predication on true, while the white circle indicates predication on false. When the `addi` instructions receive both `a` and the predicate, the instruction with the matching predicate fires and delivers the result to the subsequent shift (`slli`) instruction. Since only one `addi` instruction fires, the shift will receive only one token representing the updated value of `b`. This example is similar to the dataflow graph in Figure 1, but reduces the predicate overhead compared to partial predication by eliminating the gate and switch instructions.

Figure 2 also shows the encodings of the four instructions. The instruction fields include opcode (7 bits), predicate (2 bits), extended opcode (5 bits), immediate or target 2 (9 bits), and target 1 (9 bits). The predicate field specifies whether the instruction is predicated on a true predicate (PR = 11), predicated on a false predicate (PR = 10), or unpredicated (PR = 00). In this example, the unpredicated `teq` instruction, which has a PR field of 00, produces a "true" predicate, in which the low-order bit of the value routed to the consumers is equal to one. The `teq` has two targets, 57 and 58, which correspond to the predicate operands of the two `addi` instructions respectively. Each `addi` instruction has only one target as the second target field is needed to encode the immediate value.

3.3. Dataflow Predicate Computation

Partially predicated dataflow machines incurred overheads for the computation of compound predicates. More recent VLIW architectures provided special operations, such as wired-AND and wired-OR instructions, that permitted restricted but efficient compound predicate computation. Dataflow predication combines per-instruction predication, bipolar predicates, and implicit predication to implement efficient combining of predicates using implicit inversion, AND, and OR operators. This combination provides a solution to compound predicate computation which is equally efficient but more general than VLIW solutions, in a dataflow context.

3.4. Predicate ANDs

Figure 3a shows the C code for a simple while loop and the corresponding dataflow graph statically unrolled three times into a single TRIPS block. Each iteration consists of a load, an add, and a test instruction. Each unrolled iteration executes only if all previous unrolled while conditions evaluate to true. The compiler avoids generating compound predicates by instead predicating the test for each iteration on the test in the previous iteration. For example, the test for iteration three depends explicitly on iteration two producing a "true" predicate and implicitly on iteration one producing a "true" predicate. By predicating each `tgti` instruction on true and using the predicate generated in the previous iteration, the TRIPS compiler implements an implicit *predicate-AND* chain. This implementation is more efficient than explicit compound predicate operators because it eliminates a predicate-and instruction, reducing both code size and critical path length.

Other predicated architectures such as the HP Play-Doh [14] and IA-64 provide special instruction modes for predicate-defining instructions to reduce the height of the predicate computation tree. In particular, the wired-AND and wired-OR modes permit the conjunction or the disjunction of a limited set of predicates without the use of any additional combining instructions. Dataflow predication generalizes wired-AND and wired-OR operations, enabling an arbitrary number of predicates within a block and reducing the compute tree height for compound predicates.

3.5. Predicate ORs

The TRIPS dataflow execution model also enables optimizations that eliminate instructions common to multiple predicated paths. In Figure 3a, if the number of loop iterations is not a multiple of three, the block must terminate, executing only a fraction of the predicates and the dataflow graph. The `bro_f` instruction executes when the loop terminates, otherwise the `bro_t` performs another iteration of the loop. Whereas prior predication models needed explicit *predicate-OR* instructions to implement these tests, the TRIPS predication ISA implicitly computes the "OR" since at most one of the `tgti` instructions will produce a matching predicate. For example, if all of the `tgti` instructions produce true predicates, the `bro_f` instruction does not issue, and the loop body executes again. However, if the third unrolled iteration evaluates to false, the `bro_f` instruction receives two non-matching (true) predicates from the first two iterations, and a matching false predicate from the third iteration, and the instruction issues. By routing multiple predicates to a single instruction, the compiler can avoid generating explicit instructions to compute the compound "OR" predicate. This predicate-OR feature provides

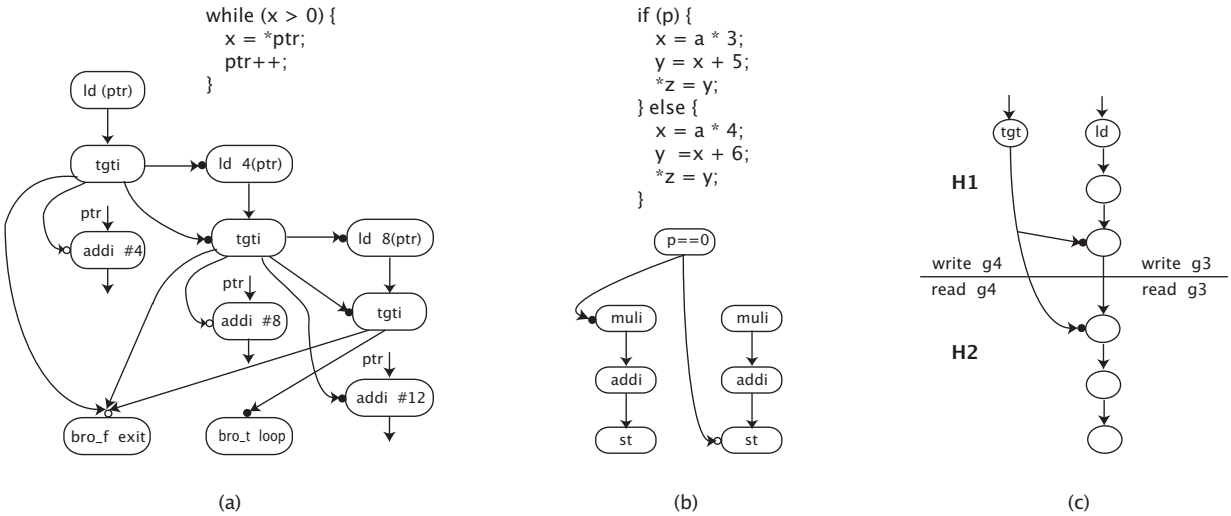

Figure 3. Example of predicate (a) computation, (b) fanout reduction, and (c) handling for long dependences.

opportunities for instruction merging as well, as shown in Section 5.

3.6. Implicit Predication

Because EDGE architectures employ direct producer/consumer bypassing instead of automatically broadcasting instruction results through a common register file, delivering a single predicate to many predicated instructions may incur significant overhead. For example, if a basic block is predicated on some predicate `p`, a naive implementation predicates every instruction in that basic block on `p`. Due to instruction size limitations, instructions that generate predicates have only one or two targets. Consequently, a naive compiler would build a software fanout tree that distributes the predicate to all the instructions, increasing dependence height and adding overhead to the block.

A dataflow predicating compiler can eliminate most of these predicates using two techniques: hoisting and implicit predication. Consider the example in Figure 3b, which has two dependence chains predicated on opposite values of a predicate `p`. In the right-hand chain, the compiler performs predication (`p` = false) at the bottom, effectively hosting the chain of instructions to execute speculatively in parallel with the computation of `p`. In the left-hand chain, the compiler inserts predication (`p` = true) at the top, thus routing the predicate to only the instruction at the top of the dependence chain and implicitly predicating the other instructions. If the predicate is non-matching (i.e., is false), the predicated instruction does not fire, so the implicitly predicated instructions in the left-hand chain will also not fire.

For implicit predication to be correct, early mispredication termination is necessary. Since implicitly predicated instructions will not fire when one of their ancestors receives a non-matching predicate, a block must be able to complete and commit even though instructions within it will never fire.

4. Microarchitectural Support

There are four areas of microarchitectural support needed for correct implementation of dataflow predication. First, the execution logic must handle dataflow predicates correctly; since predicates are not a part of the inter-block architectural state, only the execution cores and the issue logic need augmentation for basic predication. Second, support in the register files and caches can improve the performance of handling conditional register writes and stores through nullification. Third, flushing of mispredicated state once a block completes removes mispredicated dependence height as a performance consideration, and makes implicit predication possible. Fourth, the microarchitecture must raise any exceptions that occur on non-speculative paths, masking and discarding any mispredicated exceptions.

4.1 Predicated Issue Windows

The TRIPS microarchitecture employs a reservation station at each of its ALUs to hold instructions that are waiting to execute. In the reservation station, each operand requires a valid bit indicating whether the operand has arrived and a field to store the value for later use. To the dataflow execution core, a predicate operand is similar to a normal operand with one difference: its status bit tracks not whether the operand has arrived, but whether a *matching* predicate has arrived. When a value targeting a predicate operand arrives at a reservation station, the hardware logic reads the instruc-

tion's predicate bits and checks for a matching predicate. If there is a match, it updates the instruction status, but the arrival of non-matching predicates is ignored. Predication also complicates the bypassing logic since it must route an operand to one additional possible field when an operand arrives from a remote or the local execution unit.

4.2 Block Output Nullification

One possible complication with dataflow predication arises when a block output (e.g., a register write or store) occurs in one predicated path within a block but not another. Because TRIPS requires a block to produce the same outputs regardless of the path taken through the block, the compiler must generate outputs on these alternate paths. One option is to read the old value from the architectural register file or memory and write the same value back on the predicated paths that do not produce a modified value. The TRIPS ISA and microarchitecture provide an alternative, known as a `null` token, which indicates to the architectural register file or the memory system that the block has generated an output but no architectural state should be modified. The compiler inserts any necessary `null` instructions on predicated paths for this purpose. The microarchitecture includes an additional tag bit on operands indicating whether the operand is a `null` token. The ALU control logic propagates the null values through instructions until they reach a block output. This additional hardware thus eliminates the need for superfluous copies of block inputs to block outputs, at the cost of relatively little hardware complexity. Also, null tokens arriving for register writes can cause a later block to re-issue a read to that register, obtaining it from an earlier write than the nullifying block or from the architectural register file.

4.3 Early Mispredication Termination

One overhead common to predicated architectures is the expense of fetching and executing mispredicated instructions. Fetching these additional instructions may reduce the effective instruction fetch bandwidth, although this effect is mitigated since predication can eliminate expensive branch mispredictions. However, once mispredicated instructions have been fetched into the pipeline, there is no convenient way in a conventional architecture to selectively flush them. As a result, a program with a large number of mispredicated instructions that form a long sequential dependence chain will waste a large number of execution slots.

While TRIPS also incurs the overhead of fetching mispredicated instructions, the microarchitecture eliminates mispredicated instruction execution overheads. First, if the compiler predicates the tops of dependence chains, the implicitly predicated instructions on a non-matching path will not be triggered for execution as their operands will never arrive. Consequently, they will not consume any instruction execution slots or operand network bandwidth. Second, the microarchitecture need not wait until all mispredicated instructions have completed before terminating the block. Since the microarchitecture detects that a block has completed execution when it produces all of its outputs (register writes, stores, branch), it can squash long, mispredicated chains of instructions still executing in the processor core, and quickly reclaim the reservation stations for a new block. Forward progress is thus not affected by waiting for orphaned instructions to trickle through the pipeline.

4.4. Predicated Exceptions

While predication provides an opportunity for the compiler to specify speculative instruction execution, dataflow predication must preserve the exception semantics of the original program. More conventional predicated processors (such as [19]) implement a form of poison bit that the architecture sets on exceptional conditions. The poison bit later triggers an exception if the speculatively computed value becomes non-speculative. The TRIPS implementation employs a similar solution. Since TRIPS services exceptions only on a block boundary, even without predication the architecture uses a form of a poison bit called an *exception bit*. When an instruction raises an exception, the microarchitecture tags its produced operand with an exception bit, which it propagates through the dataflow graph. If the block produces any output with an exception bit set, the exception is raised and handled by the system.

The microarchitectural exception bit is necessary for correct predication. If an instruction raises an exception and its value reaches an instruction that does not receive any matching predicate, the architecture filters out the exception. These semantics are precisely the required behavior for such mis-speculative exceptions. A more complex problem arises when the predicate itself carries an exception bit. To provide well-defined execution semantics, the ISA specifies that an arriving predicate with an exception flag set is interpreted as a false predicate. If the instruction fires, it produces an exception-tagged output. Since well-formed TRIPS blocks have a single dataflow path to the output for each combination of predicates, this scheme ensures that the exception is propagated down only one predicated path of execution.

If a speculative chain of instructions lies entirely within a hyperblock, the compiler can safely predicate at the bottom of the chain as any extraneous exceptions will be filtered out prior to completion of the block. However, predicates transmitted from one hyperblock to another through writes and reads to/from the common architectural register file require special care to preserve the correct exception behavior. Fig-

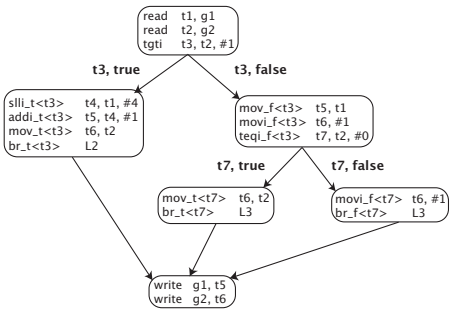

Figure 4. TRIPS block and predicate flow graph.

ure 3c shows an example in which a predicate is produced in hyperblock H_1 and used in hyperblock H_2. The speculative chain of instructions spans both hyperblocks. If the compiler predicates only the bottom of the chain in H_2, then any exception-tagged operands that are produced by speculatively executed instructions and transmitted from H_1 to H_2 would illegally trigger exceptions at the block boundaries. To ensure correct execution, any basic block split across TRIPS blocks must have every output in each hyperblock guarded by the predicate, as opposed to guarding only the outputs in the last hyperblock.

5. Compiler Predicate Optimizations

A compiler that supports dataflow predication can apply three optimizations to mitigate predication overheads. First, predicate fanout reduction removes predicates based on *intra-block* dependence chains. Second, path-sensitive predicate removal removes predicates from instructions that define *inter-block* values. Finally, a general form of instruction merging combines duplicate instructions to reduce the size of a block.

A TRIPS block is a region of the control flow graph that executes atomically and adheres to the architecturally specified block constraints [24]. In the absence of predication, a TRIPS block is simply a basic block. However, to maximize performance, the Scale compiler forms hyperblocks by combining regions of the control flow graph using if-conversion [18]. It performs all traditional loop and scalar optimizations before it forms hyperblocks. After hyperblock formation, it performs various predication optimizations, followed by global common sub-expression elimination and peephole optimization.

Scale represents hyperblocks internally as a predicate flow graph (PFG) as shown in Figure 4. A PFG is a directed graph where each edge represents a predicate and each node represents a *predicate block*–a basic block whose instructions are predicated on the incoming edges. Previous work shows that this structure allows for precise dataflow analysis in the presence of predication [3]. This phase of the compiler represents instructions in three-address form, where all intra-block communication is expressed through temporary register names. Only `read` and `write` instructions access the register file, which is used to transfer values between instructions in different blocks. The optimizations described in this section all operate on the predicate flow graph, both in and out of static single assignment form [11]. A final compiler phase schedules and translates to target form [10].

5.1. Predicate Fanout Reduction

A dataflow predicating compiler can apply both implicit predication and speculative hoisting to reduce predicate fanout, thus eliminating unnecessary predication and avoiding the insertion of move instructions that would otherwise be required to forward predicates to their consumers. For example, in Figure 4, the immediate test instruction, `tgti`, defines a predicate `t3` that predicates eight instructions (in the TRIPS ISA, immediate instructions can only target a single instruction.) In the worst case, the compiler would insert six additional move instructions to distribute the predicate to its consumers.

Scale performs predicate fanout reduction using the PFG in static single assignment form. It is free to apply implicit predication and/or hoisting to remove a predicate from any instruction, including instructions that may raise exceptions (subject to the restrictions discussed in Section 4.4). However, this strategy does not always yield the best fanout reduction. For example, if a dataflow graph has few roots and many leaves, predicating the roots offers the best fanout reduction at the possible expense of losing performance if the predicate computation is on the critical path.

The compiler removes a predicate if all of the following conditions are met: (1) the instruction is not a branch or store, (2) the instruction does not define a predicate, (3) the instruction does not define a register that is live-out of the TRIPS block, and (4) the instruction does not define a register used by a SSA ϕ-instruction. Figure 5a shows Figure 4 after the compiler performs predicate fanout reduction. It removes the predicate on the `slli` instruction, which enables the compiler to promote the instruction to its dominating predicate block and the runtime to execute the instruction regardless of the intra-block control flow.

5.2. Path-Sensitive Predicate Removal

The TRIPS ISA requires that all paths through a block produce the same set of register writes. If an instruction defines a register that is live on one path but not another, the compiler must insert additional instructions to produce a definition for the register on all paths. The compiler can either read in the register that is live and copy this value on

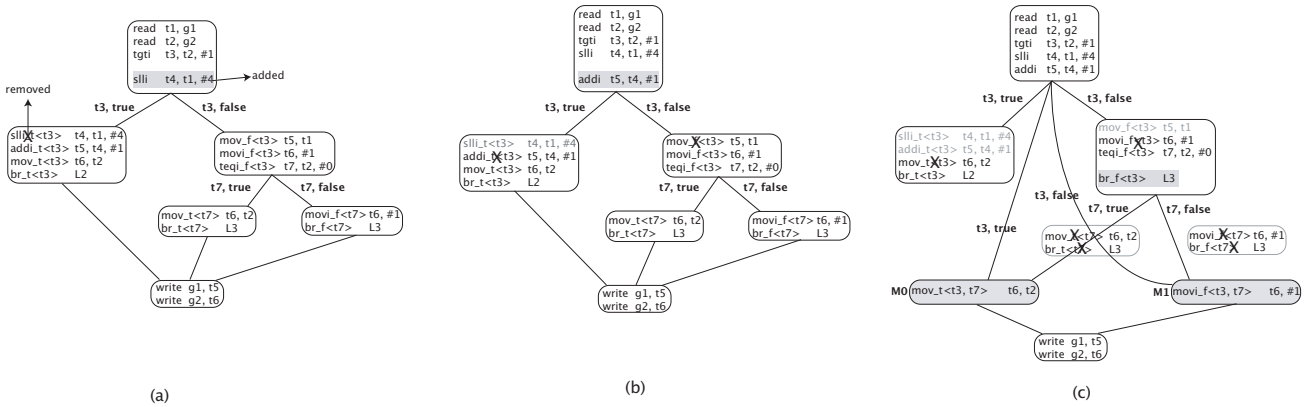

Figure 5. After (a) predicate fanout reduction, (b) path sensitive predicate removal, and (c) instruction merging.

the paths without a definition, or insert `null` instructions to nullify the write on these paths.

For example, in Figure 5a, `g1` and `g2` are both live. The compiler therefore inserts three additional `mov` instructions to write the original values of `g1` and `g2` back on the paths without the definitions. Two move instructions in the `t3-true` and `t7-true` predicate blocks set the temporary register `t6` for `g2`. One move instruction in the `t7-false` predicate block sets the temporary register `t5` for `g1`. However, the compiler does not need to preserve these registers on paths where they are not live due to multiple definitions. Path-sensitive predicate removal is an optimization that promotes instructions that define live registers to execute unconditionally. This optimization reduces the amount of predicate fanout, and also increases speculation through early resolution of inter-block dependences.

An instruction is a candidate for this optimization if: (1) the instruction defines a live register but it is not live on every path, (2) the instruction dominates the exits on which it is live, and (3) the instruction cannot raise an exception. Any candidate instruction found may be promoted to execute unconditionally. This implies that the instructions that define the candidate's operands, excluding any instructions that define predicates, must also be promoted. This recursive promotion is legal provided that no exceptions can be raised and no additional instructions (besides those in the upward dependence chain) will become speculative.

Figure 5b shows the example after path-sensitive predicate removal assuming `g1` was only live on the `t3-true` path. The `addi` instruction can be promoted to the dominating predicate block, causing it to be executed unconditionally, which enables the compiler to remove one `mov` instruction in the `t3-false` block.

5.3. Disjoint Instruction Merging

Instruction merging combines lexically equivalent instructions into a single instruction. Prior approaches require the instructions to be predicated on complementary predicates [20]. However, dataflow predication permits the merging of any lexically equivalent instructions, providing additional optimization opportunities for the compiler.

There are three categories of instruction merging: (1) instructions that have the same predicates but opposite conditions (i.e., t130 true, t130 false), (2) instructions that have different predicates but the same conditions (i.e., t130 true, t150 true), and (3) instructions that have different predicates and opposite conditions (i.e., t130 true, t150 false).

Mahlke et al. describe a straightforward method for category 1 merging [20]. The compiler identifies lexically equivalent instructions and promotes one instruction, removing the other. The compiler promotes the instruction by moving it to the block that dominates the predicate blocks containing the original instructions. For example, the two branches in the `t7-true` and `t7-false` predicate blocks shown in Figure 5b are lexically equivalent. They receive the same predicate `t7` but are predicated on opposite conditions. The compiler merges these two branches to a single branch instruction and promotes it to the dominating `t3-false` predicate block (Figure 5c.

To merge instructions that belong to category (2) and (3), the compiler exploits the predicate-OR capabilities described in Section 3. The compiler identifies instructions to merge in the same way as category (1), however it creates new instructions predicated on multiple predicates and removes the original instructions. For example, the two `mov` instructions in the `t3-true` and `t7-true` predicate blocks shown in Figure 5b are candidates for category (2) merging. The compiler eliminates these two instructions and introduces a new `mov` labeled M0 in Figure 5c. Note that this instruction is predicated on both `t3` and `t7`, exploiting the predicate-OR capability of the ISA. At runtime, at most one of the predicates will be true, obeying the predication rules. Similarly, the compiler merges the `mov` instructions in the `t3-false` and `t7-false` predicate blocks into a single instruction.

a) Loop extracted from genalg

```
for (x = c;
   rx > 0.0 && x < pop-1;
   x++, p_fitness++)
      rx -= *p_fitness;
```

b) Instruction sequence for loop body

```
lw      t4, (t3) L[0]
fstod   t5, t4
fsub    t6, t0, t5     ; rx  -= *p_fitness
addi    t7, t1, 1      ; x++
addi    t8, t3, 4      ; p_fitness++
fgt     t9, t6, t100
tlt_t<t9>      t10, t7, t2
bro_t<t10>     genalg$4      ; loop back
bro_f<t9>      genalg$5      ; loop exit
bro_f<t10>     genalg$5      ; loop exit
```

c) Predicate guards for live-outs

```
mov_f<t9>     rx, t6
mov_f<t10>    rx, t6
mov_f<t9>     x,  t7
mov_f<t10>    x,  t7
mov_f<t9>     pf, t8
mov_f<t10>    pf, t8
```

d) After predicate combining

```
movi_f<t9, t10>  tmp, 1
mov_f<tmp>       rx, t6
mov_f<tmp>       x,  t7
mov_f<tmp>       pf, t8
```

Figure 6. Instruction merging in genalg.

The compiler merges category (3) instructions by flipping the condition for the test instruction that generates the predicate, and then applying the category (2) transformation.

The principal benefit from instruction merging within a hyperblock is that it eliminates instructions, creating space for the inclusion of more useful instructions into the hyperblock. Such inclusion exposes opportunities for additional compiler optimization. The example depicted in Figure 5 reduces the size of a TRIPS block by three instructions—six instructions eliminated and three new ones added.

Figure 6 shows an instruction merging example using a kernel snippet extracted from *genalg*, a genetic algorithm application developed by MIT Lincoln Laboratories. The top left portion of the figure depicts the source code, while the bottom left portion shows the equivalent instructions for the loop body, not including the register read and write instructions. In this example, `x`, `rx`, and `p_fitness` are all live past the loop. The test instructions (`fgt` and `tlt`) represent the predicate-AND chain required for implementing the short-circuiting loop condition checks. As described in Section 3, such an and-chain ensures correct exception semantics.

If the loop executes for several iterations, statically unrolling to fill the 128-instruction block with as many iterations as possible maximizes the parallelism found in the instruction window. This potentially high degree of Unrolling exposes many opportunities for instruction merging. An example merges the two exiting branch instructionsof the first iteration. The compiler applies category (2) merging to form a single branch instruction. Likewise, the move instructions generating each of the live registers are also candidates for merging.

Instruction merging also enables predicate fanout reduction. As shown in Figure 6c, the loop exit predicates for the first iteration (`t9, t10`) control each of the three live register values. Since each instruction can encode at most two targets and there are four consumers for the predicate (including the branch), so a total of four fanout instructions are required for the two predicates. It the compiler merges the two predicates as shown in Figure 6d and sends the resulting predicate to the register producing instructions, then it eliminates three fanout instructions—no fanout instructions are required for `t9` and `t10` and one fanout instruction is required for `tmp`.

Performing these optimizations by hand, we unrolled several iterations of the *genalg* loop to maximally fill a block. Compared to the best performing compiler, these optimizations improved the performance by over 2.25 times.

6. Performance Analysis

We evaluate predicate fanout reduction and path-sensitive predicate removal using 28 EEMBC 2.0 benchmarks, the Scale compiler with a TRIPS back end, and a cycle-accurate simulator called *tsim-proc*. This simulator closely models the TRIPS prototype microarchitecture [8]—a recent performance evaluation estimated that tsim-proc and an RTL-level simulator differ by less that 4% on average using a set of microbenchmarks. Tsim-proc faithfully models most delays in the prototype implementation, including a one-cycle hop from any tile to an adjacent tile, a 32KB, 2-way set-associative distributed L1 data cache with a 2-cycle latency, a 64KB, 2-way set-associative distributed L1 instruction cache with a 1-cycle latency, an 8-cycle block fetch latency, and a 3-cycle branch prediction latency.

Figure 7 shows the results of various compiler optimizations. We present the results in three categories, all of which use hyperblocks—*intra*: predicate fanout reduction, *inter*: path-sensitive predicate removal, and *both*: both predicate fanout reduction and path-sensitive predicate removal. We use hyperblocks with no predicate optimizations as a baseline. We also show results for basic blocks only (no predication) as *BB*. We do not include instruction merging, as we are still in the process of adding the optimization into the Scale compiler.

Predicate fanout reduction obtains an average speedup of 11% over hyperblocks alone. We observe an average 14% reduction in dynamic move instructions, a 2% reduction in total dynamic instructions and a 5% reduction in the number of dynamic blocks. Path-sensitive predicate removal has an average speedup of 1%, but several benchmarks stand out as benefiting from this optimization including autcor00, conven00 and iirflt01, with speedups of 5-9%. In these benchmarks, instruction promotion indirectly benefits both the branch predictor and the load-store dependence predictor by enabling early resolution of branches and stores, resulting in higher prediction accuracy and improved performance. Combining the two optimizations produces an average speedup of 12%. The benchmark rotate01 shows the

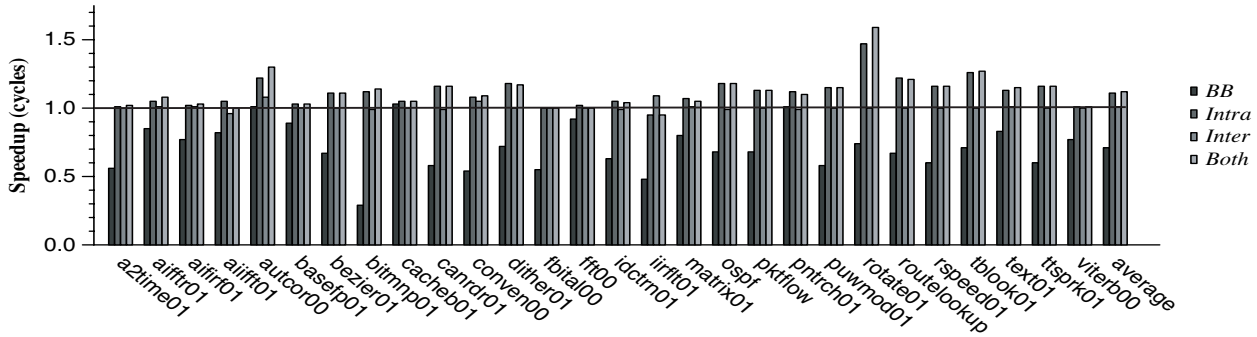

Figure 7. Speedup on EEMBC Embedded Benchmark Suite.

most marked improvement, having a combined speedup of 59%. Basic blocks are on average 29% slower than hyperblocks alone and 41% slower than hyperblocks with both predicate optimizations. The benefits of optimized predication are multifold: increased instruction window utilization, reduction in the number of blocks (static) executed, reduced branch mispredictions, and improved I-cache performance.

7. Conclusions

Dataflow predication exploits ISA features, microarchitectural mechanisms, and compiler algorithms to reduce predication overheads in an EDGE ISA while maintaining low-complexity out-of-order issue. In VLIW architectures, the execution overhead of falsely predicated instructions limits the compiler's ability to perform aggressive predication. In superscalar architectures, the hardware complexity and ISA encoding difficulties inhibit the incorporation of full predication. Dataflow predication avoids both of these limitations, while reducing the predicate encoding space consumed to two bits per instruction. However, dataflow predication incurs the costs of fanning out predicates to many consumers. The following components of dataflow predication reduce its overhead and provide opportunities for improved performance:

- *Computation of compound dataflow predicates:* By supporting dual-polarity predicates, predicated test instructions, and receipt of multiple non-matching predicates per instruction (predicate-ORing), the ISA reduces the overhead of computing compound predicates.
- *Implicit predication and early mispredication termination:* The microarchitecture supports removal of blocks while falsely predicated instructions are executing speculatively. This capability is necessary to support implicit predication, which, along with speculative hoisting, can significantly reduce the number of consumers to which predicates must be sent using software fanout trees.
- *Disjoint instruction merging:* Since the ISA supports multiple sources for an instruction's predicate, the compiler can merge instructions with distinct predicates, eliminating redundant instructions.

We presented a preliminary evaluation of the dataflow predication optimizations in the TRIPS architecture. Predicated hyperblocks improve performance by 29%, on average, over basic blocks. Fanout minimizations both within and across multiple hyperblocks improve performance by an additional 12%. Although instruction merging in the Scale compiler is still immature, we demonstrated that, with hand merging of instructions, significant speedups were achievable on several benchmarks.

In the near term, there are two additional optimizations that are likely to further reduce the overheads of dataflow predication. First, a short-circuiting *and* instruction, that follows C semantics, will permit tree-based computation of predicate chains and reduce the predicate dependence height. Second, predicate multicast operations that trade instruction placement flexibility for the ability to route a predicate to more consumers using shorter, wider fanout trees will reduce dependence heights and instruction overheads while improving performance.

In the longer term, the biggest remaining overheads of predication may eventually be the fraction of mispredicated instructions in the window, which reduce the effective window size. At any given moment in a program's execution, there are three classes of instructions in the window– useful instructions that are correctly predicated, useless instructions that are falsely predicated, and instructions past a branch misprediction, all of which are useless. Each window size has a sweet spot between no predication (pure superscalar) and all predication (pure dataflow) for maximum parallelism. If instruction window sizes continue to increase, however, the relative costs of increased predication will continue to decline, pushing the ideal balance toward more aggressive predication. It is possible that the long-term solution to branch mispredictions will not be more accurate predictors, but conversion of most unpredictable branches to predicates in extremely large instruction windows. For this solution to be viable, some form of predicate predication [9] will likely be necessary to reduce the increases in dependence heights, caused by predication, down the true paths of execution.

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-1892 and NBCH30390004, NSF instrumentation grant EIA-0303609, NSF CAREER grants CCR-9985109 and CCR-9984336, IBM University Partnership awards, and grants from both the Alfred P. Sloan Foundation and the Intel Research Council.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[2] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[3] D. I. August. *Systematic compilation for predicated execution*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[4] D. I. August, J. W. Hwu, J.-M. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu. The program decision logic approach to predicated execution. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 208–219, May 1999.

[5] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 92–103, Dec. 1997.

[6] M. Beck, R. Johnson, and K. Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, 1991.

[7] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, October 2004.

[8] 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 the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[9] W. Chuang and B. Calder. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 183–192, June 2003.

[10] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2006.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[12] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 26–38, April 1989.

[13] J. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

[14] V. Kathail, M. Schlansker, and B. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

[15] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[16] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, 2005.

[17] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[18] B. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[19] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 238–247, October 1992.

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

[21] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129, Apr. 1994.

[22] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towie. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs. *Computer*, 22(1):12–26, 28–30, 32–35, January 1989.

[23] R. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[24] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *Fourth International IEEE/ACM Symposium on Code Generation and Optimization (CGO)*, pages 185–195, March 2006.

[25] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, December 2003.

[26] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report TR-370, LCS, MIT, Cambridge, MA, August 1986.

[27] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 15–25, January 2001.

[28] N. J. Warter, D. M. Lavery, and W. W. Hwu. The benefit of predicated execution for software pipelining. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 497–506, January 1993.