How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures

Behnam Robatmili[1] Dong Li[2] Hadi Esmaeilzadeh[3] Sibi Govindan[4]
Aaron Smith[5] Andrew Putnam[5] Doug Burger[5] Stephen W. Keckler[2,6]
[1]Qualcomm Research Silicon Valley [2]University of Texas at Austin
[3]University of Washington [4]AMD [5]Microsoft Research [6]NVIDIA
behnamr@qti.qualcomm.com

Abstract

Dynamic multicore architectures, that fuse and split cores at run time, potentially offer a level of performance/energy agility that static multicore designs cannot achieve. Conventional ISAs, however, have scalability limits to fusion. EDGE-based designs offer greater scalability but to date have been performance limited by significant microarchitectural bottlenecks. This paper addresses these issues and makes three major contributions. First, it proposes Iterative Path Prediction to address low next block prediction accuracy and low speculation rates. It achieves close to taken/not-taken prediction accuracy for multi-exit instruction blocks while also speculating the predicated execution path within the block. Second, the paper proposes Exposed Operand Broadcasts to address the overhead of operand delivery for high fanout instructions by exposing a small number of broadcast operands in the ISA. Third, we present a scalable composable architecture called T3 that uses these mechanisms and show it can operate across a wide range of power and performance spectrum by increasing energy efficiency and performance significantly. Compared to previous EDGE designs, T3 improves energy efficiency by about 2x and performance by up to 50%.

1 Introduction

Traditional power scaling methods such as dynamic voltage and frequency scaling (DVFS) are becoming less effective given the current trends of transistor scaling [28, 9]. One possible direction is to use architectural innovations to distribute execution of each thread across a variable number of processing cores in a flexible manner [9, 28, 11, 8, 12]. Such dynamic distributed microarchitectures can operate at different energy and performance points, supplementing traditional DVFS methods. Explicit Data Graph Execution (EDGE) [22] architectures were conceived with the goals of offering high performance, high energy efficiency, and high flexibility, by distributing computation across many simple tiles. By raising the level of control abstraction to an atomic, predicated, multi-exit block of instructions, in which branches are converted to predicates, control overheads such as branch prediction and commit can be amortized. By incorporating dataflow semantics into the ISA, aggressive out-of-order execution is possible while using less energy than out-of-order RISC or CISC designs. The intra-block data-flow encodings push much of the run-time dependence graph construction to the compiler, reducing the energy required to support out-of-order execution through construction and traversal of those graphs. To date, EDGE architectures have not yet demonstrated these potential advantages as a result of two main weaknesses [7].

The first weakness is associated with predicates. Unlike out-of-order designs that use predicates to avoid hard-to-predict branches [4, 16], EDGE architectures employ predicates to build large instruction blocks to reduce fetch bottlenecks. The combination of speculative block-based execution and predication within blocks in EDGE architectures moves branch prediction off of the critical path and alleviates the fetch bandwidth bottleneck. However, performing multi-exit, next-block prediction on each block results in loss of prediction accuracy as the global history of branches no longer includes those branches that have been converted into predicates. Additionally, the branches that are converted to predicates are evaluated at the execution stage rather than being predicted, thus manifesting themselves as execution bottlenecks. This paper proposes a new distributed predictor organization called an Iterative Path Predictor (IPP). IPP quickly predicts an approximate predicate path through each block and uses it to speculatively execute high-confidence predicates in the block once those instructions are fetched. It also uses that predicted predicate path to predict the next-block target address by appending the path to the global history. IPP increases the rate of predictions (improving performance) and both inter- and intra-block prediction accuracy (saving energy). IPP improves performance by 15% and yields a 5% energy saving with 16 composed cores per thread.

The second weakness in prior EDGE designs is high-fanout operand delivery. For low-fanout operations, using dataflow communication among a block's instructions eliminates the need for the broadcast bypass network, associative tag matching, and register renaming logic found in conventional out-of-order processors. However, for high-fanout operands, our baseline EDGE compiler generates trees of move instructions to propagate values to destination instructions. These fanout instructions increase execution delay and consume additional energy. This paper proposes an ISA extension called Exposed Operand Broadcasts (EOBs). EOBs, which are low-power broadcast tags, are assigned by the compiler to the highest-fanout operands. Different from register tags, EOBs are not assigned dynamically and do not require centralized power-hungry register renaming. They also consume little bypass energy, as their bit width is small. Using 16 composed cores, EOBs result in a speedup of 5% and 10% energy savings.

This paper makes three main contributions. First, IPP resolves the scalability issues of control dependences and speculative execution across many cores. Second, EOBs handle data dependences between instructions in a scalable and power efficient manner. Third, we demonstrate that a scalable composable core architecture, including mechanisms that reduce the negative effects of control and data dependences can address a much wider range of power and performance than traditional fixed-core architectures that support only DVFS. We implement IPP and EOB solutions along with other recently proposed mechanisms [18, 19] in a new microarchitecture (with ISA extensions) called T3. T3 improves performance and energy-delay product by about 50% and 2x, respectively, compared to TFlex [12], a previously proposed EDGE architecture.

2 Background

EDGE ISAs [22] were designed with the goals of high single-thread performance, ability to run on a distributed, tiled execution substrate, and good energy efficiency. An EDGE compiler converts program code into single-entry, multiple-exit predicated blocks. The two main features of an EDGE ISA are block-atomic execution [15] and static dataflow [5] within a block. Instructions in each block use dataflow encoding through which each instruction directly encodes its destination instructions. Using predication, all intra-block branches are converted to dataflow instructions. Therefore, within a block, all dependences other than memory are direct data dependences. An EDGE ISA uses architectural registers and memory for inter-block communication. This hybrid dataflow execution model supports efficient out-of-order execution, conceptually using less energy to construct the dependence graphs, but still supports conventional languages and sequential memory semantics. Each block is logically fetched, executed, and committed as a single atomic entity. This block-atomic execution model amortizes the book-keeping overheads across a large number of instructions and reduces the number of branch predictions and register accesses. It also reduces the frequency of control decisions, providing the latency tolerance needed to make distributed execution across multiple cores practical.

The TRIPS microarchitecture implemented the first instantiation of an EDGE architecture. TRIPS supports fixed-size EDGE blocks of up to 128 instructions, with 32 loads or stores per block. Instructions could have one or two dataflow targets, and instructions with more than two consumers in a block employed `move` instructions, inserted by the compiler to fan operands out to multiple targets. To achieve fully distributed execution, the TRIPS microarchitecture uses no global wires, but was organized as a set of replicated tiles communicating on routed networks. The TRIPS design has a number of serious performance bottlenecks [7]. Misprediction flushes are particularly expensive because the TRIPS next-block predictor has low accuracy compared to modern predictors, and the refill time for such a large window was significant. Since each instruction block is distributed among the 16 execution tiles, intra-block operand communication is energy- and latency-expensive. The predicates used for intra-block control cause performance losses, as they are evaluated in the execution stage, but would have been predicted as branches in conventional superscalar processors. Finally, the registers and data caches distributed around the edges of the execution array limit register and memory bandwidth, forcing some instructions to have long routing paths to access them.

TFlex was a second-generation EDGE microarchitecture [12], which implemented the TRIPS ISA but improved upon the original TRIPS microarchitecture. TFlex distributes the memory system and control logic, making each tile a fully functional EDGE core, but permits a dynamically determined number of tiles to cooperate on executing a single thread. Thus, TFlex is a dynamic multicore design, similar in spirit to CoreFusion [11]. The ability to run a thread on a varied number of cores, from one to 32, is a major improvement over TRIPS, which has fixed execution granularity. Due to this fixed granularity, TRIPS is unable to adapt the processing resources in response to changing workload mix, application parallelism, or energy efficiency requirements. Unlike the registers, instruction cache banks, and data cache banks that TRIPS distributes along the edges of the execution array, the TFlex microarchitecture interleaves and distributes these storage structures across all participating cores to facilitate better scalability and bandwidth. TFlex distributes the control responsibilities across all participating cores by employing distributed protocols to implement next-block prediction, fetch, commit, and misprediction recovery without centralized logic, enabling the architecture to scale to 32 participating cores per thread. Each TFlex core has the minimum

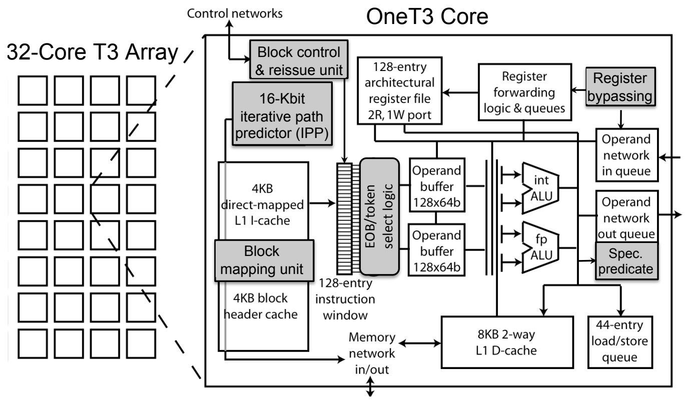


Figure 1. T3 Block Diagram.

resources required for running a single block, including a 128-entry RAM-based instruction queue, a L1 data cache bank, a register file, a branch prediction table, and an instruction (block) cache bank. When *N* cores are composed, they can run *N* blocks simultaneously, of which one block is non-speculative. Similar to TRIPS, TFlex design distributes the instructions from each in-flight block among all participating cores, increasing operand communication latency. TFlex also has the software fanout trees, poor next-block prediction accuracy, and no speculation on predicates.

The T3 microarchitecture addresses bottlenecks in TFlex, including speculation accuracy and operand delivery. Figure 1 shows the T3 microarchitecture block diagram with shaded boxes representing the new components designed for performance and energy efficiency. T3 employs a new predictor design called an Iterative Path Predictor (IPP – described in Section 3), which unifies branch target and predicate prediction while providing improved accuracy for each. On the other hand, instead of solely relying on intra-block dataflow mechanisms to communicate intra-block operands, T3 employs ISA-exposed operand broadcast operations (EOBs – explained in Section 4). In addition to IPP and EOBs, T3 employs other mechanisms for further improving power efficiency. To reduce high intra-block communication, *deep block mapping* [18] maps each block to the instruction queue of one core, permitting all instructions to execute and communicate within the core. *Critical inter-block value bypassing* [19] bypasses remote register forwarding units by sending late-arriving register values directly from producing to consuming cores. *Block reissue* [19] permits previously executed instances of a block to be reissued while they are still in the instruction queue, even if they have been flushed, thus saving energy.

3 Iterative Path Predictor

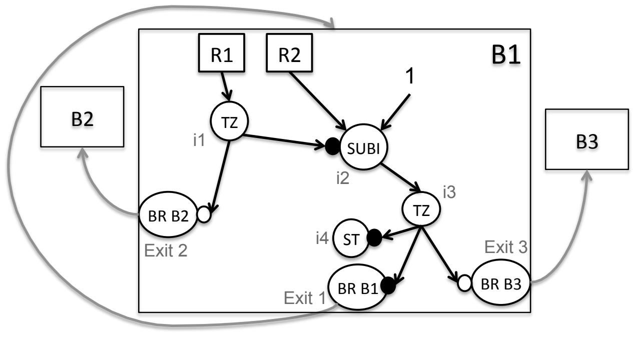
An EDGE compiler uses predication to generate large multi-exit blocks by converting multiple nested branches into predicates. Therefore, all control points within a block are converted into a DAG of predicates generated by dataflow test instructions. By speculatively executing several of these large predicated dataflow blocks, EDGE microarchitectures can reduce fetch, prediction, and execution overheads, and can distribute single-thread code across light-weight cores. In these architectures, instead of predicting each single branch instruction, prediction is performed on a block-granularity using a *next block predictor* or *target predictor*. This predictor predicts the next block that will be fetched following the current block. As EDGE blocks can have multiple exits, each block can have multiple next block addresses depending on the history of the previously executed blocks and the execution path within the block determined by the predicates. Figure 2 shows a sample code, its dataflow representation, and a diagram corresponding to the predicated dataflow block of the code. In the dataflow representation, the target fields of each instruction represent a destination instruction and the type of the target. For example, p and $op1$ represent the *predicate* and *first operand* target types, respectively. The two branches in the original code (I_1 and I_3) are converted to dataflow test instructions (i_1 and i_3). During execution, once a test instruction executes, its predicate value (1 or 0) is sent to the consuming instructions of that test instruction. The small circles in the diagram indicate the predicate consumer instructions and their predicate polarity. The white and black circles indicate the instructions predicated on true and false, respectively. For instance, the `subi` only executes if the i_1

```
I1:  bz R1, B2
I2:  subi a, R2, 1
I3:  bz a, B3
I4:  ST ADDR
I5:  j B1
```

(a) Initial representation

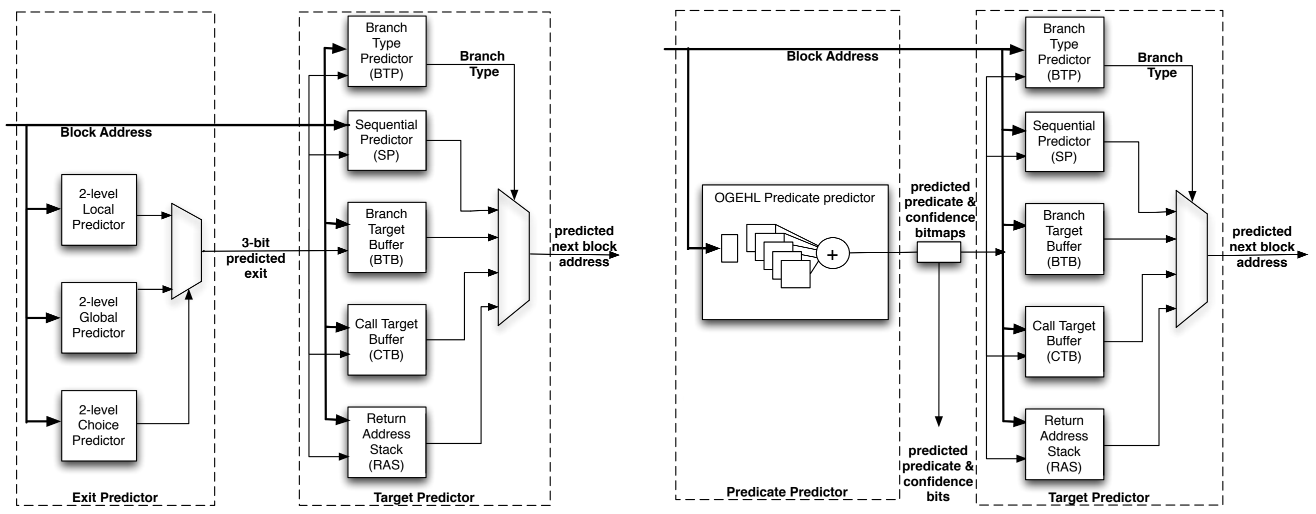
```
Read R1 <i1,op1>
Read R2 <i2,op1>
i1:  tz <e2,p><i2,p>
i2:  subi_f 1 <i3,op1>
i3:  tz <e1,p> <e3,p> <i4,p>
i4:  ST_f ADDR
e1:  br_f B1
e2:  br_t B2
e3:  br_t B3
```

(b) Dataflow representation



(c) Dataflow diagram

Figure 2. Sample code, its equivalent predicated dataflow representation and predicated dataflow block including two predicated execution paths and three possible exits.


Block Address
2-level Local Predictor
2-level Global Predictor
2-level Choice Predictor
Exit Predictor
3-bit predicted exit
Branch Type Predictor (BTP)
Sequential Predictor (SP)
Branch Target Buffer (BTB)
Call Target Buffer (CTB)
Return Address Stack (RAS)
Branch Type
predicted next block address
Target Predictor
Block Address
OGEHL Predicate predictor
+
predicted predicate & confidence bitmaps
predicted predicate & confidence bits
Predicate Predictor
Branch Type Predictor (BTP)
Sequential Predictor (SP)
Branch Target Buffer (BTB)
Call Target Buffer (CTB)
Return Address Stack (RAS)
Branch Type
predicted next block address
Target Predictor

(a) TFlex next block predictor

(b) Iterative path predictor (IPP)

Figure 3. Block diagram of TFlex block predictor and Iterative Path Predictor.

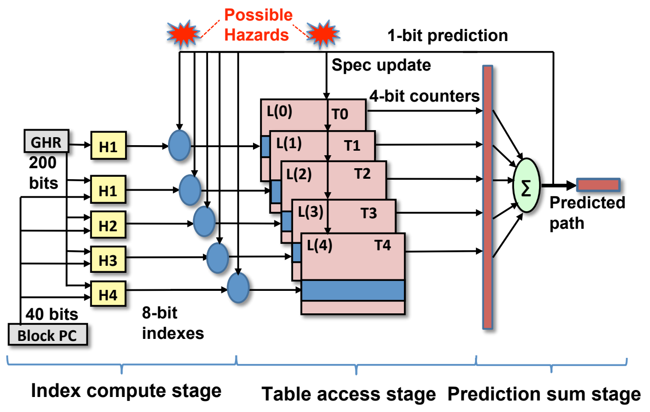
test instruction evaluates to zero (*false*). Depending on the value of the predicate instructions, this block takes one of three possible exits. If i_1 evaluates to 1, the next block will be block $B2$. If both i_1 and i_3 evaluate to 0, this block loops back to itself (block $B1$). Finally, if i_1 and i_3 evaluate to 0 and 1, this block branches to block $B3$. This model of predicated execution changes the control speculation problem from one-bit taken/not-taken prediction to multi-bit predicate path prediction when fetching each block. Thus, an accurate EDGE predictor must use a global history of the predicates in previous blocks to predict the predicate path that will execute in the current block and then use that information to predict the next block. This section proposes the first such predictor called Iterative Path Predictor (IPP).

One drawback associated with predicated dataflow blocks is that the test instructions producing the predicates within blocks are executed and not predicted like normal branches. A critical path analysis showed that when running SPEC benchmarks across 16 composed TFlex cores, on average 50% of the critical cycles belong to instructions waiting for predicates. In Figure 2(b), i_1 will not execute until the value of $R1$ has arrived. Similarly, i_3 will not execute until both $R1$ and $R2$ have arrived and the result of the i_2 (`subi`) instruction is evaluated. To mitigate this bottleneck caused by intra-block predicates, IPP uses the predicted predicate path of each block to speculate on the value of predicates within that block, thus increasing the speculation rate among the distributed cores.

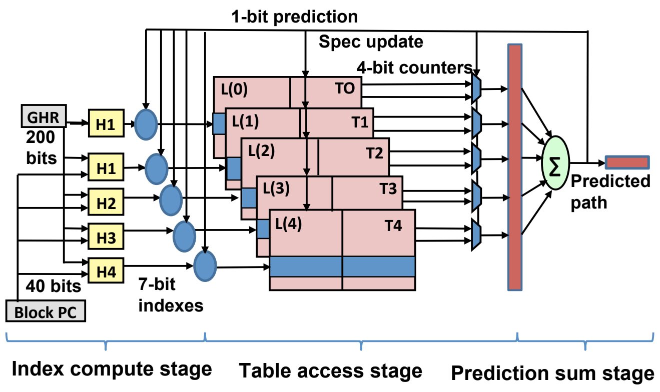
3.1 Fused Predicate/Branch Predictor

Previous EDGE microarchitectures predict the block exit to perform next block prediction. The original TFlex predictor is distributed across all participating cores. For each block, next block prediction is performed by a core selected based on the block PC, regardless of the core executing that block. The predictor core and executing core communicate the prediction and the prediction results during the fetch/commit/flush of the block. Figure 3(a) illustrates the block diagram of the next block predictor in each TFlex core. This 16K-bit predictor consists of two 8K-bit components: (a) an exit predictor that is an Alpha 21264-like tournament predictor that predicts a three-bit exit code (the ISA allows between one and eight unique exits from each block) of the current block, and (b) a target predictor that uses the predicted exit code and the current block address to predict the next block address (PC). Because each exit can result from a different branch type, the target predictor supports various types of targets such as *sequential, branch, call*, and *return* targets. For the block shown in Figure 2(c), the TFlex exit predictor predicts which of the three exits from the block ($Exit$ 1 to 3) will be taken and then the target predictor maps the predicted exit value to one of the target block addresses ($B1$ to $B3$ in the figure).

Similar to the TFlex predictor, IPP is a fully distributed predictor with portions of prediction tables distributed across participating cores. IPP uses the TFlex mapping mechanisms to identify which core holds the needed tables for each block. Figure 3(b) shows the block diagram of the IPP predictor in each core. Instead of predicting the exit code of the current block, IPP contains a predicate predictor that iteratively predicts the values of the predicates (predicate paths) in the current block. The predicted values are grouped together as a predicted *predicate bitmap* in which each bit represents a predicate in the block. For example, for the block shown in Figure 2(c), the bitmap will have two bits with the first and second bits predicting the results



(a) Pipelined OGEHL predictor



(b) Hazard-free pipelined OGEHL predictor

Figure 4. Two OGEHL-based pipeline design used in IPP.

of the test instructions i_1 and i_3. The target predictor is similar to the target predictor used by the TFlex block predictor. It uses the predicted predicate bits (values) along with the block address to predict the target of the block. The rest of this subsection discusses the predicate predictor in IPP.

Predicting predicates in each block is challenging since the number of predicates per block is not known at prediction time. For simplicity, the predicate predictor used by IPP assumes a fixed number of predicates in each block. The predicate predictor component must predict multiple predicate values as quickly as possible so that it does not become the system bottleneck. After studying different predictors, we designed an optimized geometric history length (OGEHL) predictor [23] for predicate value (path) speculation. The original OGEHL branch predictor predicts each branch in three steps. First, in the *hash compute* step, the branch address is hashed with the contents of the global history register (GHR) using multiple hash functions. Then, the produced hash values are used to index multiple prediction tables in the *table access* step. Each entry in these tables is a signed saturating counter. Finally, in the *prediction* step, the sum of the indexed counters in the prediction tables is calculated and its sign is used to perform prediction. Positive and negative correspond to taken and not-taken branches or true and false predicate values, respectively. The absolute value of the sum is the estimated confidence level of the prediction. By comparing the confidence level to a threshold, a confidence bit is generated for each prediction. When the prediction is performed, the corresponding counters in the tables and the GHR value are updated speculatively. We use the best reported OGEHL predictor in [23] with eight tables and a 200-bit global history register (modified from the original 125-bit GHR). Assuming this best-performing predictor distributed across 16 cores, the size of the prediction tables stored on each core is about 8Kbits, which is equal to the size of the exit predictor in the original TFlex predictor shown in Figure 3(a). Therefore, using IPP does not incur any additional area overhead. When a core performs a next block prediction, it broadcasts its changes to the GHR to other cores to keep the global history registers consistent across cores.

To accelerate the predicate path prediction, we optimize the OGEHL predictor by converting each step in the OGEGL predictor into a pipeline stage as shown in Figure 4(a). Although, this predictor can predict one predicate in each cycle, this pipeline may have data hazards when predicting back-to-back dependent predicates in one block. For example, if the second predicate in a block is *false* only when the first predicate is *true*, this correlation is not captured in this pipeline because when the first prediction is still in flight, in the $prediction$ stage, the second prediction is in the $access$ stage. To address this issue, a hazard-free pipelined OGEHL shown in Figure 4(b) reads dual prediction values from each prediction table in the table access stage. The correct value is selected at the end of that stage depending on the prediction value computed in the $prediction$ stage (selecting the second prediction based on the first prediction). Using this pipelined OGEHL predictor may introduce some energy overhead, for which we account in our energy measurements. However, our results indicate that this energy overhead is negligible compared to the energy savings due to improved next block prediction accuracy and predicate prediction.

3.2 Speculative Execution of Predicates

When the next target of a block is predicted, the predictor sends the predicted predicate bitmap to the core executing that block. It also sends another bitmap called the *confidence bitmap* with each bit representing the confidence of its corresponding predicted predicate. When an executing core receives the predication and confidence bitmaps,

```
I1:    add c, a, b
I2:    sub e, c, d
I3:    add f, c, g
I4:    bz x L1
I5:    st c, f
I5a:   j EXIT
L1:
I6:    st e, f
```

(a) Initial representation

```
i1:    add <i2, op1> <i1a, op1>
i1a:   mov <i3, op1> <i5 op1>
i2:    sub <i6, op1>
i3:    add <i5, op2> <i6, op2>
i4:    testnz <i5, pred><i6, pred>
i5:    st_t
i6:    st_f
```

(b) Dataflow representation

```
i1:    add [S-EOB=1, op1]
i2:    sub [R-EOB=1] <i6, op1>
i3:    add [R-EOB=1] <i5, op2><i6, op2>
i4:    testnz <i5, pred><i6, pred>
i5:    st_t [R-EOB=1]
i6:    st_f
```

(c) Dataflow/EOB representation

Figure 5. A sample code and corresponding code conversions for the hybrid dataflow/EOB model.

it stores the information required for speculative execution of the predicates in the instruction queue. The instruction queue is extended to contain one *confidence* bit and one *prediction* bit for each predicate-generating test instruction. For each predicate with its confidence bit set (meaning high prediction confidence), the speculation starts immediately after receiving these bits by sending the predicted value to its destination instructions. For example, assume the bitmap associated with the block shown in Figure 2(c) is `00`, meaning that the i_1 and i_3 predicates are both predicted to be 0. If both confidence bits are also set, the *store* instruction, i_4, is executed and the block loops through $Exit1$ immediately, thus avoiding waiting for predicates to be computed and input registers R1 and R2 to arrive. If the bitmap is `10` or `11`, $Exit2$ is taken, ignoring all instructions in the block and branching directly to block $B2$. For detecting predicate misspeculations, this mechanism relies on the dataflow execution model used by TFlex. The speculated test instructions in a block still receive their inputs values from other instructions inside the block. Once all inputs of a speculated test instruction have arrived, that instruction executes, its output is compared against the predicted value of that predicate and if the two do not match, a misspeculation flag is raised. Consequently, the block and all of the blocks that depend on it are flushed from the pipeline and the prediction tables are updated for that block. That block is then fetched and re-executed in the *no-predicate-prediction* mode.

4 ISA-Exposed Operand Broadcasts

By eliminating register renaming, result broadcast, and associative tag matching in the instruction queue, the direct dataflow intra-block communication achieves substantial energy savings for low-fanout operands compared to conventional out-of-order designs. However, the energy savings are limited in the case of high-fanout instructions for which the compiler needs to generate software fanout trees [7]. Each instruction in the EDGE ISA can encode up to two destinations. As a result, if an instruction has a fanout of more than two, the compiler inserts *move* instructions to form a dataflow fanout tree for operand delivery. Previous work [7] has shown that for the SPEC benchmarks, 25% of all instructions are *move* instructions. These fanout *move* trees manifest at runtime in the form of extra power consumption and execution delay. To alleviate this issue, this paper proposes a novel *hybrid* operand delivery that exploits compile-time analysis to minimize both the delay and energy overhead of operand delivery. This mechanism uses direct dataflow communication for low-fanout operands and compiler-generated ISA-exposed operand broadcasts (EOBs) for high-fanout operands. These limited EOBs eliminate most of the fanout overhead of the move instructions providing both performance and power improvements by (1) eliminating fetch and execution of these instructions, and (2) executing fewer code blocks because block formation is more efficient without the excess instructions.

4.1 Static EOB Assignment

The original EDGE compiler [25] generates blocks containing instructions in dataflow format in which each instruction directly specifies up to two of its consumers using a 9-bit instruction identifier for each target. For instructions with more consumers than targets, the compiler builds fanout trees using *move* instructions. The modified EOB-enabled compiler accomplishes two additional tasks: choosing which high-fanout instructions should be selected for one of the limited intra-block EOBs, and assigning one of these EOBs to each selected instruction. The number of available EOBs is determined by a microarchitectural parameter called $MaxEOB$. The compiler uses a greedy algorithm, sorting all instructions in a block with more than two targets and selecting instructions to broadcast based on the number of targets. Starting from the beginning of the list, the compiler assigns each instruction in the list an EOB from the fixed number of available EOBs until all EOBs are assigned or the list is empty. The compiler must encode the EOB in both the producer and consumer instructions. Each instruction can produce up to one *Send* EOB and consume up to two *Receive* EOBs.

Figure 5 illustrates a sample program, its equivalent dataflow representation, and its equivalent hybrid dataflow/EOB representation generated by the modified compiler. In Figure 5(a), *a, b, d, g,* and *x* are the inputs read from registers and except for *stores*, the first operand of each instruction is the destination. In the dataflow code shown in Figure 5(b), instruction i_1 can only encode two of its three targets. Therefore, the compiler inserts a *move* instruction, instruction i_{1a}, to generate the fanout tree for that instruction. For the hybrid communication model shown in Figure 5(c), the compiler assigns an EOB (1 in this example) to i_1, the instruction with high fanout, and encodes the broadcast information into both i_1 and its consuming instructions (instructions i_2, i_3 and i_5). Finally, the compiler uses dataflow direct communication for the remaining low-fanout instructions, e.g. instruction i_2.

4.2 Microarchitectural Support for EOBs

Superscalar cores broadcast every operand to every instruction, which requires a wide CAM. EOBs use the ISA to greatly reduce the number of broadcasts, so that only the small number of instructions that need to send a broadcast communicate with the instructions waiting for a broadcast operand. This approach greatly reduces the number of sends, receives, and the width of the associative logic needed to match senders and receivers. Although they both use CAMs, EOBs are more energy efficient than the instruction communication model in superscalar processors for several reasons. First, because EOBs use small identifiers, the bit width of the CAM is small compared to a superscalar design which must track a larger number of renameable physical registers. Second, the compiler can select which instruction operands are broadcast, which in practice is a small fraction of the total instruction count. Third, only a portion of instructions in the queue are broadcast receivers and perform an EOB comparison during each broadcast. To implement EOBs in T3 cores, a small *EOB CAM* array stores the receive EOBs of broadcast receiver instructions in the instruction queue.

Figure 6 illustrates the instruction queue of a single core when running the broadcast instruction i_1 in the sample code shown in Figure 5(c). When the broadcast instruction executes, its send EOB (value *001* in this example) is sent to be compared against all of the potential broadcast receiver instructions in the instruction queue. Only a subset of instructions in the instruction queue are broadcast receivers, while the rest need no EOB comparison. Operands that have already received their broadcast do not have to perform CAM matches, saving further energy. Upon an *EOB CAM* match, the hardware generates a write-enable signal to write the operand into the instruction queue entry of the corresponding receiver instruction. The broadcast type field of the sender instruction (*operand 1* in this example) is used to select the column containing the receivers. Tag delivery and operand delivery do not happen on the same cycle. Similar to superscalar operand delivery networks, the EOB of the executing sender instruction is first delivered one cycle before instruction execution completes. On the next cycle, when the result of the broadcast instruction is ready, its output is written simultaneously into all matching operand buffers in the instruction window. Figure 6 also illustrates a sample circuit implementation for the compare logic in each *EOB CAM* entry. The CAM tag size in this figure is three bits which represents the bit width of $EOBs$. In this circuit, the compare logic is disabled if one of the following conditions is true: (1) if the instruction corresponding to the CAM entry has been previously issued; (2) if the receive EOB of the instruction corresponding to the CAM entry is not valid, which means the instruction is not a broadcast receiver (for example instruction i_5 in Figures 5 and 6); or (3) if the executed instruction is not a broadcast sender.

5 Results

5.1 Experimental Methodology

We use an execution-driven, cycle-accurate simulator to simulate the TRIPS, TFlex, and T3 processors [12]. The simulator is validated against the cycles collected from the TRIPS prototype chip. In TFlex or T3 modes, the simulator supports different configurations in which a single thread can run across a number of cores ranging from 1 to 16 cores in powers of 2. We limit the number of composed cores between 1 and 16 as performance and power scaling does not improve much when merging more than 16 cores for integer benchmarks. The power model uses CACTI [27] models for all major structures such as instruction and data caches, SRAM arrays, register arrays, branch predictor tables, load-store queue CAMs, and on-chip network router FIFOs to obtain a per-access energy for each structure. Combined with access counts from the architectural simulator, these per-access energies provide the energy dissipated in these structures. The power models for integer and floating point ALUs are derived from both Wattch [2] and the TRIPS hardware design database. The combinational logic power in various microarchitectural units is modeled based on detailed gate and parasitic capacitances extracted from RTL models and activity factor estimates from the simulator. The baseline EDGE power models at 130nm are suitably scaled down to 45nm using linear technology scaling. We use a supply voltage of 1.1 Volts and a core frequency of 2.4 GHz for the TRIPS, TFlex, and T3 platforms. We accurately model the delay of each optimization used by the T3 simulator. Also, we use CACTI and scaled TRIPS power models to estimate the power consumed by the components used by various T3 features, including the OGEHL tables and the EOB CAM and comparators.

To examine the performance/power flexibility of the T3 microarchitecture, we compare it to several design points in the performance and power spectrum of production processors. We use the Intel *Core 2* and *Atom* as representatives for high performance and lower power platforms respectively, and rely on the chip power and performance measurement results reported in [6] for these platforms at the same 45nm technology node. We use the McPAT [14] models to estimate the core power consumption to compare against T3. The main idea of such a comparison is not a detailed, head-to-head comparison of T3 to these plat-

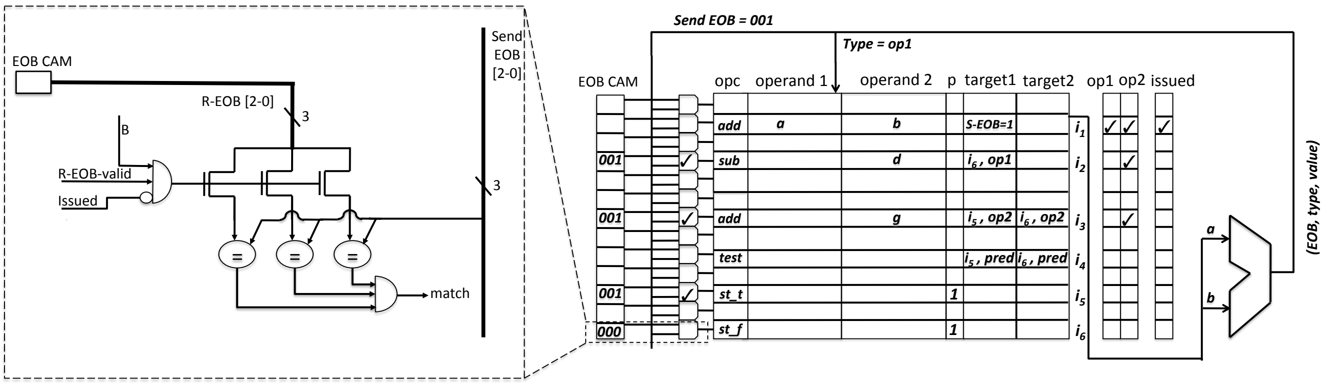


Figure 6. EOB CAM compare logic (left) and broadcast instruction execution (right).

	TFlex original predictor	Basic pipelined IPP	Hazard-free pipelined IPP
Block prediction MPKI	4.03	3.29	2.93
Predicate prediction MPKI	N/A	0.65	0.54
Average speedup	1.0	1.11	1.14

Table 1. Pedictor accuracy and speedup for SPEC INT.

forms, but to demonstrate the power/performance flexibility offered by T3 in the context of these other chips. The benchmarks include 15 SPEC 2000 benchmarks (7 integer and 8 floating point) each simulated with a single *Simpoint* [24] region of 100 million instructions as well as 25 EEMBC benchmarks executed to completion. We exclude Fortran and C++ SPEC benchmarks as they are not supported by the TRIPS compiler. Since SPEC and EEMBC results are very similar, we only show the SPEC results in this paper.

Next, the section presents a power/performance design space exploration of IPP and EOBs. To illustrate the power and performance scalability of IPP and EOBs, the section then compares the fully integrated T3 system to previous EDGE microarchitectures across different core composition granularities and microarchitectural features.

5.2 Design Exploration for IPP

Table 1 compares different proposed pipelined IPP designs including the basic pipelined IPP and the hazard-free pipelined IPP shown in Figures 4 for SPEC INT benchmarks. The prediction accuracy of FP benchmarks is significantly higher and does not affect the analysis. In this experiment, each SPEC benchmark runs using 16 composed cores. This table presents MPKI (mispredictions per kilo instructions) for both next block prediction and predicate value speculation. It also presents speedups compared to the original TFlex predictor show in Figure 3(a). Using the basic pipelined IPP improves next block prediction MPKI from 4.03 to 3.29. By capturing the correlation between consecutive predicates in each block, the hazard-free pipeline improves MPKI to 2.93, while improving predicate prediction MPKI from 0.65 down to 0.54. Of the 14% speedup achieved by the hazard-free IPP pipeline, the contributions of speculative execution of predicates and improved next block prediction accuracy are 12% and 2%, respectively. This predictor increases core-level energy consumption by 1.2%, most of which is consumed by the OGEHL adders. However, energy saved by this predictor because of the improved next block and predicate prediction accuracy is 6%, resulting in a total energy saving of 4.8%. This energy saving can be significantly improved by using top predication in which predicates are placed on top of dependent chains of instructions, instead of bottom predication used in this study. Using IPP to predict top predicates results in the same performance boost while preventing useless execution of instructions on incorrect predicate paths.

Table 2 evaluates the hazard-free IPP design when varying the number of predicted predicate values per block. The next block prediction accuracy first improves when increasing predicted branches (predicate values) from 1 to 3 and then degrades. This observation is supported by the fact that for most SPEC benchmarks, the average number of executed predicates per block is three. The predicate prediction MPKI, however, increases consistently as the number of speculated predicates increases from 1 to 5, which has a minor effect on performance. While the best next block prediction is achieved when predicting three predicates per block, the best speedup occurs when predicting 4 predicates per block due to the increased intra-block speculation.

5.3 Design Exploration for EOBs

By converting high fanout operands in instructions with at least three targets, the compiler eliminates the fanout trees. However, the overall energy benefit is dependent on the total number of available EOBs. Increasing the number of the available EOBs ($MaxEOBs$) reduces the operand delivery energy until the overheads from EOB width be-

Number of predicted predicates per block	1	2	3	4	5
Block prediction MPKI	4.43	4.00	2.86	2.93	2.96
Predicate prediction MPKI	0.10	0.29	0.44	0.54	0.57
Ave. speedup vs. TFlex	1.03	1.04	1.12	1.14	1.13

Table 2. Accuracy and speedups of the pipelined IPP when varying the number of predicted predicates per block for SPEC INT.

comes dominant. Figure 7 illustrates the energy breakdown into executed *move* and broadcast instructions for a variety of $MaxEOBs$ values on the SPEC benchmarks each running across 16 composed cores. The energy values are normalized to the total energy consumed by *move* instructions when instructions within each block communicate only using dataflow ($MaxEOBs$ = 0). When only using dataflow (the original TFlex operand delivery), all energy overheads are caused by the *move* instructions. Allowing one or two broadcast operations in each block, $MaxEOBs$ of 1 and 2, we observe a sharp reduction in the energy consumed by *move* instructions. The compiler chooses the instructions with highest fanout first when assigning EOBs. For these $MaxEOBs$ values, the energy consumed by EOBs is low. As we increase the total number of EOBs, the energy consumed by broadcast operations increases dramatically and fewer move instructions are removed. At 16 EOBs, the broadcast energy becomes dominant. For high numbers of $MaxEOBs$, the broadcast energy is an order of magnitude larger than the energy consumed by *move* instructions. The key observation in this graph is that allowing only 4 to 8 broadcasts in each block minimizes the total energy consumed by *moves* and broadcasts. For such $MaxEOBs$, the total overhead energy is about 28% lower than the energy consumed by the baseline TFlex ($MaxEOBs = 0$) and about 2.7x lower than when $MaxEOBs$ is equal to 128. We also note that for $MaxEOBs$ larger than 32, the energy consumed by *move* instructions is at a minimum and does not change, but the *EOB CAM* becomes wider so the energy consumed by EOBs continues growing.

Using 3-bit EOBs removes 73% of dataflow fanout instructions and instead 8% of all instructions are encoded as the EOB senders. These instructions send EOBs to 34% of instructions (EOB receivers). Using 3-bit EOBs results in about 10% total energy reduction on T3 cores. The consumed energy is reduced in two ways: (1) it saves the energy consumed during execution of the fanout trees which constitute more than 24% of all instructions; and (2) by better utilizing the instruction blocks, it reduces the fetch and decode operations by executing 5% fewer blocks. Also, EOBs improve performance in all benchmarks except *vpr*. When using EOBs for this benchmark, memory instructions are receiving their operands sooner and firing in a different order; thus causing more violations in load/store queues.

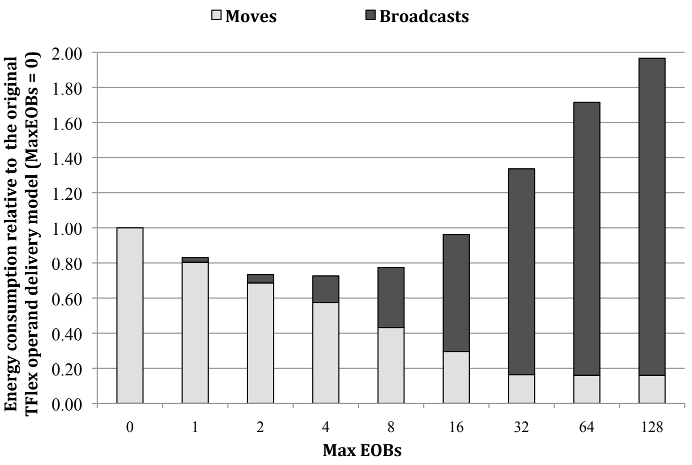
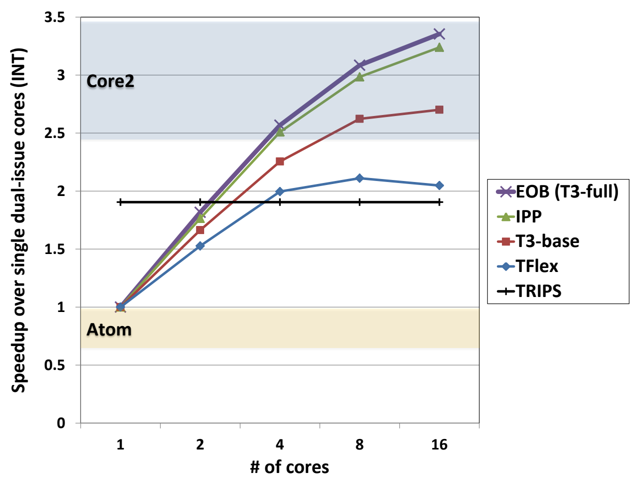


Figure 7. Averaged energy breakdown between move instructions and broadcasts for various numbers of available EOBs.

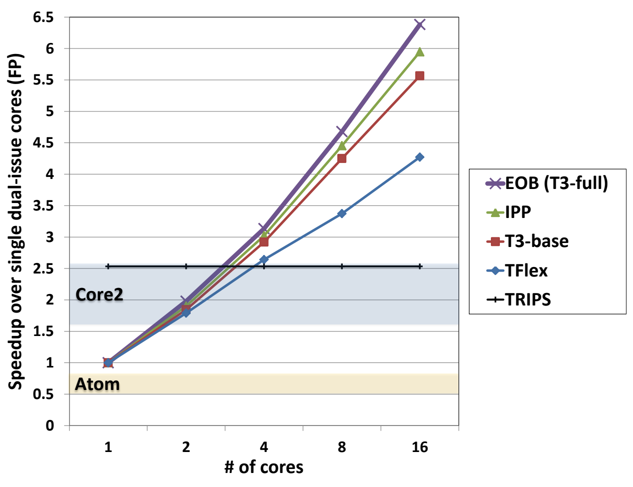
5.4 Performance and Energy Scalability

This subsection compares efficiency and scalability of IPP and EOBs and the full T3 system against several previously proposed original and enhanced EDGE architectures. Figure 8 shows the average speedup and energy consumption (L2 energy excluded) for (a) *TRIPS*; (b) *TFlex* [12]; (c) *T3*-base which is a system that applies previously proposed EDGE optimizations including *deep mapping* [18], *block reissue* [19] and *register bypassing* [19] on top of TFlex; and (d) *T3-full* which is a fully integrated T3, that applies EOBs and IPP on top of T3-base. T3-base is the same as T3-full without IPP and EOBs. These results are normalized to a common baseline, which is TFlex with one core. Since the 1-core baseline runs only one block, it does not support speculation and IPP. However, T3 1-core can support EOBs to become slightly faster than TFlex 1-core. We did not enable EOBs in T3 1-core to simplify the graphs. This figure also breaks down the contributions of IPP and EOBs on the full T3 system (IPP and EOB charts). The EOBs used in these experiments are 3 bits wide and IPP uses the hazard-free pipeline predicting up to 4 predicates per block. In these graphs, T3 and TFlex charts are reported in different configurations each running different core counts ranging from 1 to 16. TRIPS results are straight lines since it does not support composability.

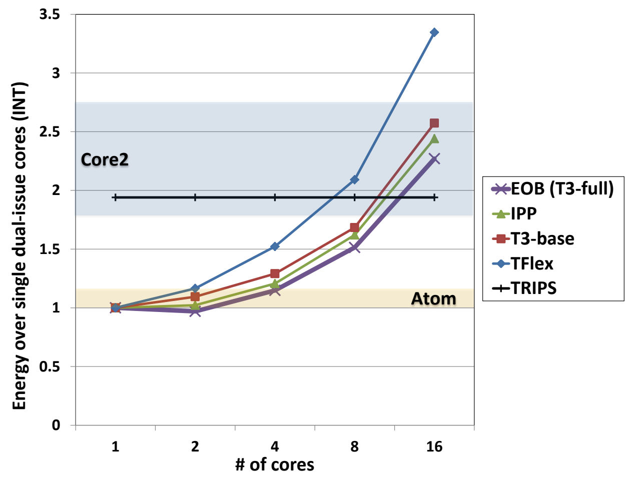
T3 illustrates a significant reduction in consumed energy and increase in performance compared to both TRIPS and TFlex. This major increase in energy efficiency is largely attributed to the IPP and EOBs. For INT benchmarks, Figures 8(a) and 8(c) show that *TFlex-8* (TFlex using 8 cores) outperforms TRIPS by 1.12× while consuming slightly more energy. However, relying on the optimized microarchitectural components, *T3-8* (EOB charts with 8 cores in the figure), outperforms TRIPS by 1.43× while consuming about 25% less energy. *T3-4* achieves the best



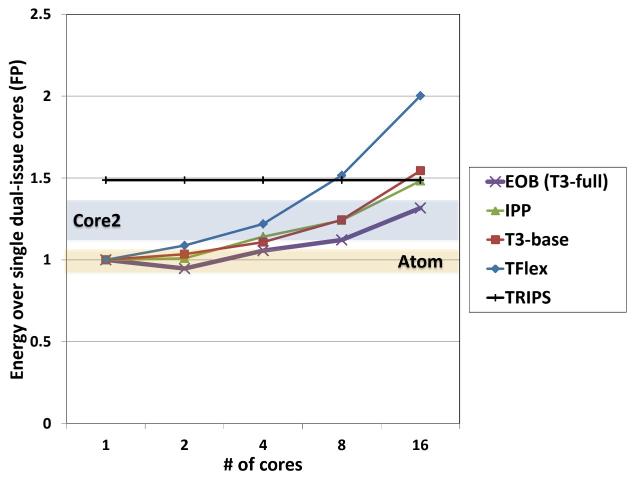
(a) SPEC INT Speedup



(b) SPEC FP Speedup



(c) SPEC INT Energy



(d) SPEC FP Energy

Figure 8. Average speedups and energy over single core for SPEC with varying numbers of composed cores and optimizations (shaded areas represent Core2 and Atom DVFS operational ranges).

inverse-energy-delay-product (inverse-EDP). This value is 1.8× of that of *original TFlex* and 2.6× of that of TRIPS. More than half of this increase caused by the combination of IPP and EOBs. For FP benchmarks, *TFlex-16* outperforms TRIPS by about 1.7× while consuming 30% more energy. *T3-16* (EOB charts), on the other hand, outperforms TRIPS by about 2.5× while consuming 1.1× less energy. *T3-16* reaches the best energy efficiency in terms of inverse energy-delay-product (inverse-EDP) and inverse-ED^2P which are 3.2× and 7× better compared to TRIPS.

To better quantify power and performance benefits of IPP and EOBs in the T3 system, we focus on the speed and power breakdown for INT benchmarks, which are inherently hard for a compiler to parallelize automatically. On average, *T3-16* outperforms *TFlex-16* by about 1.5× across both INT and FP benchmarks, which translates to a speedup of about 50%. For the INT benchmarks, the speedups stem primarily from the IPP (14%) compared to other high-performing optimizations such as deep block mapping (7%), and block reissue (10%). As shown in the energy graphs, the T3 optimized cores save significant energy compared to original TFlex. For example *T3-16* consumes about 35% less energy than *TFlex-16* for SPEC INT benchmarks. The main energy saving results from EOBs (11%) compared to other energy-saving optimizations such as deep block mapping (8%), and block reissue (7%).

T3 can span a broad range of points in the energy/performance spectrum beyond what conventional processors can achieve using DVFS. Figure 8 also reports relative performance and energy results of *Atom* and *Core 2*. In each graph, different voltage and frequency operating points of *Core 2* represent the high-performance operating region (2.4GHz/1.1v and 1.6GHz/0.9v). Similarly, operating points of *Atom* represents the low-energy operating region (1.6GHz/1.1v and 800MHz/0.8v). T3 runs only vary the number of composed cores with a fixed frequency and voltage equal to that of the high *Core 2* voltage and frequency operating point (2.4GHz/1.1v). T3 achieves high energy efficiency in both low-energy and high-performance regions. By composing only a few T3 optimized cores, T3 can achieve major performance boosts in low-energy regimes. For example, while the energy consumed by *T3-2* falls within the low-energy region (Figures 8(c) and 8(d)), its performance is close to the range of the high-performance region (Figures 8(a) and 8(b)). Merging more cores significantly boosts performance at a relatively small energy cost. For example, while *T3-4* and *T3-8* perform at or above the high-performance region, their consumed energy is below this region. While conventional processors are typically limited in their energy/performance space by possible DVFS configurations, T3's composable cores enable a different axis on which to trade performance and energy. As composibility is independent of DVFS, the combination of the two techniques can further extend the range of power/performance trade-offs. For instance, 1, 2, 4, 8, or 16 composed cores with 5 DVFS points provides 25 different energy-efficient operating points in the power/performance spectrum as opposed to 5 with DVFS alone.

6 Related Work and Generality

The most related studies fall into three main categories:

Distributed architectures. WiDGET [28] decouples thread context management units from execution units and can adapt resources to operate in different power-performance operating points. CoreFusion [11] is a fully dynamic approach that fuses up to 4 cores with a conventional ISA using central control and renaming units. T3, on the other hand, exploits distributed control and execution mechanisms such as IPP that use no central control unit. Similar mechanisms may be employed by other distributed architectures such as CoreFusion [11] or WiDGET [28]. For example, with a mechanism similar to IPP, it is possible to support trace caches in a distributed architecture. In such an IPP-like design, for a given trace address, a dedicated core can include the next trace prediction data. A local predictor in that core can use the shared GHR to predict the next trace address. Previously proposed mechanisms for trace caches such as Path-based (or trace paths) predictors [21] and multiple-branch predictors [20] do not exploit OGEHL and are not distributable. They use an extended Pattern History Table and a central GShare predictor.

Predicate prediction. The predicate prediction mechanisms in the literature [4, 16, 13] only focus on lightly predicated code, in which dependent chains of predicates do not exist. IPP proposed in this paper, conversely, performs both branch and predicate prediction across large and heavily predicated hyperblocks, where chains of dependent predicates exist. The insights from the IPP design can be adopted by any design that supports predication, enabling more aggressive predication to streamline the fetch and better utilize fetch bandwidth.

Hybrid operand delivery. Most previous architectures have relied on either broadcast or fanout exclusively. The most related work [3, 10, 17] used hardware to select the right mechanism dynamically, which is less precise than the compiler and includes hardware runtime overhead that the EOBs do not incur. Our approach can be applied directly to any other explicitly token-based systems, such as dataflow architectures, WaveScalar [26], or Transport-Triggered Architecture [1]. With ISA extensions, a ForwardFlow architecture [8] can also take advantage of this mechanism. ForwardFlow dynamically generates an explicit internal dataflow representation to scale instruction wakeup, selection, and issue across multiple cores. An EOB-like compile time analysis can help ForwardFlow save power by not generating these graphs at runtime and also by focusing only on critical low-fanout data dependences.

7 Conclusions

This paper demonstrates an energy-scalable composable multicore architecture can provide a much wider dynamic range of energy efficiency and performance than both conventional and previous EDGE designs have achieved. To achieve a high degree of energy efficiency and scalability, this paper addresses two fundamental issues associated with composable block-based dataflow execution. The Iterative Path Predictor solves the low multi-exit next bock prediction accuracy and low speculation rate due to heavy predicate execution. The Exposed Operand Broadcasts reduce the energy consumed and latency incurred by compiler-generated trees of move instructions built for wide-fanout operands. Exploiting both low-overhead architecturally exposed broadcasts and direct dataflow communication, the proposed architecture (called T3) supports fast and energy-efficient operand delivery for high- and low-fanout instructions. Exploiting these mechanisms, T3 removes operand delivery and speculation bottlenecks and improves performance and energy efficiency by about 50% and 2x, respectively, compared to prior EDGE designs. Such a design achieves high energy efficiency at different power and performance operating points across a wide power/performance spectrum and extends the power/performance tradeoffs beyond what conventional processors can offer using traditional voltage and frequency scaling. These features make such composable designs an attractive candidate to be used in systems employed for a wide range of workloads under varying power and performance constraints.

Acknowledgements

We thank Mark Gebhart, Jeff Diamond and Karin Strauss for their useful feedback. This work was supported in part by National Science Foundation grant CCF-0916745 and DARPA contract F33615-03-C-4106.

References

[1] K. 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.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *ACM SIGARCH Computer Architecture News*, 28(2):83–94, June 2000.

[3] R. Canal and A. González. A Low-complexity Issue Logic. In *International Conference on Supercomputing*, pages 327–335, May 2000.

[4] W. Chuang and B. Calder. Predicate Prediction for Efficient Out-of-order Execution. In *International Conference on Supercomputing*, pages 183–192, June 2003.

[5] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Data-flow Processor. In *International Symposium on Computer Architecture*, pages 126–132, December 1975.

[6] H. Esmaeilzadeh, T. Cao, X. Yang, S. Blackburn, and K. McKinley. Looking Back on the Language and Hardware Revolution: Measured Power Performance, and Scaling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–332, March 2011.

[7] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. V. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An Evaluation of the TRIPS Computer System. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, March 2009.

[8] D. Gibson and D. A. Wood. Forwardflow: A scalable core for power-constrained CMPs. In *International Symposium on Computer Architecture*, pages 14–25, June 2010.

[9] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Transactions on Computers*, 41(7):33–38, July 2008.

[10] M. Huang, J. Renau, and J. Torrellas. Energy-efficient Hybrid Wakeup Logic. In *International Symposium on Low Power Electronics and Design*, pages 196–201, August 2002.

[11] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, June 2007.

[12] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *International Symposium on Microarchitecture*, pages 381–394, December 2007.

[13] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In *International Symposium on Microarchitecture*, pages 43–54, November 2005.

[14] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture*, pages 469–480, December 2009.

[15] S. Melvin and Y. Patt. Enhancing Instruction Scheduling With a Block-Structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, June 1995.

[16] E. Quinones, J.-M. Parcerisa, and A. Gonzalez. Improving Branch Prediction and Predicated Execution in Out-of-Order Processors. In *International Symposium on High Performance Computer Architecture*, pages 75–84, February 2007.

[17] M. A. Ramírez, A. Cristal, A. V. Veidenbaum, L. Villa, and M. Valero. Direct Instruction Wakeup for Out-of-Order Processors. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 2–9, January 2004.

[18] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for Mapping Dataflow Blocks to Distributed Hardware. In *International Symposium on Microarchitecture*, pages 23–34, November 2008.

[19] B. Robatmili, M. S. S. Govindan, D. Burger, and S. Keckler. Exploiting criticality to reduce bottlenecks in distributed uniprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 431–442, December 2011.

[20] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *International Symposium on Microarchitecture*, pages 24–34, December 1996.

[21] E. Rotenberg, J. E. Smith, E. Rotenberg, and J. E. Smith. Path-based Next Trace Prediction. In *International Symposium on Microarchitecture*, pages 14–23, December 1997.

[22] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *International Symposium on Computer Architecture*, pages 422–433, June 2003.

[23] A. Seznec. The O-GEHL Branch Predictor. *Instruction-Level Parallelism Special Issue: The 1st JILP Championship Branch Prediction Competition*, December 2004.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, June 2001.

[25] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE Architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, 2006.

[26] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. WaveScalar. In *International Symposium on Microarchitecture*, pages 291–302, December 2003.

[27] D. Tarjan, S. Thoziyoor, and N. Jouppi. HPL-2006-86, HP Laboratories, Technical Report. June 2006.

[28] Y. Watanabe, J. D. Davis, and D. A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *International Symposium on Computer Architecture*, pages 2–13, June 2010.