# Compiler-assisted Hybrid Operand Communication

Dong Li        Behnam Robatmili,        Madhu Saravana Sibi Govindan        Aaron Smith
Steve Keckler                                Doug Burger

*University of Texas at Austin, Computer Science Department*
{dongli , beroy, sibi, asmith, skeckler, dburger}@cs.utexas.edu

## ABSTRACT

Communication of operands among in-flight instructions can be power intensive, especially in superscalar processors where all result tags are broadcast to a small number of consumers through a multi-entry CAM. Token-based point-to-point communication of operands in dataflow architectures is highly efficient when each produced token has only one consumer, but inefficient when there are many consumers due to the construction of software fanout trees. Placing operands in registers is efficient for broadcasting the values which have consumers spread over a long lifetime, but inefficient for shorter-lived operations. This paper evaluates a compiler-assisted hybrid instruction communication model that combine tokens instruction communication with statically assigned broadcast tags. Each fixed-size block of code is given a small number of architectural broadcast identifiers, which the compiler can assign to producers that have many consumers. Producers with few consumers rely on point-to-point communication through tokens. Producers whose result is live past the instruction block communicate with distant consumers through a register. Selecting the mechanism statically by the compiler relieves the hardware from categorizing instructions at runtime. At the same time, a compiler can categorize instructions better than dynamic selection does because the compiler analyzes a larger range of instructions. Furthermore, compiler could perform complex optimizations without hardware cost and execution-time penalty. We propose a compiler optimization to reuse broadcast tags for instructions with non-overlapping broadcast live ranges, the speedup is further improved without spending more power . The results show that this compiler-assisted hybrid token/broadcast model requires only eight architectural broadcasts per block, enabling highly efficient CAMs. This hybrid model reduces instruction communication energy by 28% compared to a strictly token-based dataflow model (and by over 2.7X compared to a hybrid model without compiler support), while simultaneously increasing performance by 8% on average across the SPECINT and EEMBC benchmarks, running as single threads on 16 composed, dual-issue EDGE cores.

## 1. INTRODUCTION

Communicating operands between instructions is a major source of energy consumption in modern processors. A wide variety of operand communication mechanisms have been employed by different architectures. For example in superscalar processors, to wake up all consumer instructions of a completing instruction, physical register tags are broadcast to power-hungry Content Addressable Memories (CAMs), and operands are obtained from a complex bypass network or by a register file with many ports. A mechanism commonly used for operand communication in dataflow architectures is point-to-point communication, which we will refer to as

"tokens" in this paper. Tokens are highly efficient when a producing instruction has a single consumer; the operand is directly routed to the consumer, often just requiring a random-access write into the consumer's reservation station. If the producer has many consumers, however, dataflow implementations typically build an inefficient software fanout tree of operand-propagating instructions (that we call *move* instructions). These two mechanisms are efficient under different scenarios: broadcasts should be used when there are many consumers currently in flight (meaning they are in the instruction window), tokens should be used when there are few consumers, and registers should be used to hold values when the consumers are not yet present in the instruction window.

Several approaches [5, 6, 12, 18, 19] have proposed hybrid schemes which dynamically combine broadcasts and tokens to reduce the energy consumed by the operand bypass. These approaches achieve significant energy consumption compared to superscalar architectures. In addition, because of their dynamic-nature, these approaches can adapt to the window size and program characteristics without changing the ISA. On the other hand, these approaches use some additional hardware structures and keep track of various mechanisms at runtime.

The best communication mechanism for an instruction depends on the dependence patterns between that instruction and the group of consumer instructions currently in the instruction window. This information can be calculated statically at compile time and conveyed to the microarchitecture through unused bit in the ISA.

Using this observation, this paper evaluates a compiler-assisted hybrid instruction communication mechanism that augments a token-based instruction communication model with a small number of architecturally exposed broadcasts within the instruction window. A narrow CAM allows high-fanout instructions to send their operands to their multiple consumers, but only unissued instructions waiting for an architecturally specified broadcast actually perform the CAM matches. The other instructions in the instruction window do not participate in the tag matching, thus saving energy. All other instructions, which have low-fanout, rely on the point-to-point token communication model. The determination of which instructions use tokens and which use broadcasts is made statically by the compiler and is communicated to the hardware via the ISA. As a result, this method does not require instruction dependence detection and instruction categorization at runtime. In addition, the compiler can reduce the bit width of the needed CAM by efficiently reusing the tags for non-overlapping live range broadcasts. However, this approach requires ISA support and may not automatically adapt to microarchitectural components such as window size.

Table 1 categorizes different architectures by their operand communication mechanisms and instruction dependences detection methods. Architectures in the first row use the software (compiler) to

detect instruction dependences. However, architectures in the second row use hardware to detect instruction dependences. Architectures in each column use a specific mechanism for instruction communication. Superscalar and dataflow mechanisms each represent different ends of this spectrum. Superscalar processors use hardware just for broadcast, which provides high sharing between consumer. Dataflow machines, on the other hands use compiler just for point-to-point communication, which provides no sharing between consumers. Dynamic hybrid schemes [5, 6, 12, 18, 19] use hardware to detect instruction dependences and dynamically select the right communication mechanism for each instruction. Forwardflow [11] dynamically uses only point-to-point communication for all instructions. Finally, the approach discussed in this paper (referred to as Compiler-assisted broadcast/tokens in the table) uses software to detect dependences and categorize instructions for token-based or broadcast communication mechanisms.

|  | Broadcast | Hybrid | Tokens |
|---|---|---|---|
| SW |  | **Compiler-assisted broadcast/tokens** | Dataflow [8, 3] TRIPS [21] Wavescalar [24] |
| HW | Superscalar | N-use table [5] Hybrid Wakeup [12] | Forwardflow [11] |

**Table 1: Different operand delivery approaches categorized by operand communication mechanisms used and instruction dependency detection methods.**

Our experimental vehicle is TFlex [13], a composable multicore processor, which implements an EDGE ISA [21]. We extend the existing token-based communication mechanism of TFlex with this hybrid approach and evaluate the benefits both in terms of performance and energy. On a composed 16-core TFlex system (running in the single-threaded mode), the proposed compiler-assisted hybrid shows a modest performance boost and significant energy savings over the token-only baseline (which has no static broadcast support). Across the SPECINT2K and EEMBC benchmarks, using only eight architectural broadcasts per block, performance increases by 8% on average. Energy savings are more significant, however, with a 28% lower energy consumption in operand communication compared to the token-only baseline. This energy saving translates to a factor of 2.7 lower than a similar hybrid policy implementation without full compiler support.

## 2. STRATEGIES FOR OPERAND COMMUNICATION

Providing efficient operand communication between instructions is essential for performance and power efficiency in modern processors. Different mechanisms for instruction communication have been used by different architectures. Superscalar processors use fully-dynamic, power-hungry broadcast bypass networks for handling instruction dependences that exist in the instruction window. For other dependence patterns, they use registers and memory. Dataflow processors, on the other hand, use static point-to-point communication (also referred to as tokens) among producer and consumer instructions. To be able to select the right communication mechanism or a combination of mechanisms in a given system, one must understand the tradeoff space of instruction communication. Hence, we first discusses this tradeoff space in this section, and then we introduce different communication mechanisms. For each mechanism, we also explain its advantages and disadvantages.

### 2.1 Operand Bypassing Tradeoff Space

Several parameters are important when designing operand communication mechanisms among producing and consuming instructions. One of the most important of these parameters is *fanout*, which is defined as the number of consuming instruction for a given producer. Other key parameters are *dependence distance*, which is defined as the distance between each producer and its consumers in the number of dynamic instructions, and the *instruction window size* that affects the number of producers and consumers currently in flight. Figure 1 plots the density of instruction communication with different fanout and dependence distances. We collected this data using an Alpha ISA simulator [17] for the SPEC2000 integer benchmarks [2]. The z-axis represents the normalized communication density, so upper parts of the graph represent a higher density of communication. Assuming inorder fetch, the x-axis measures the dynamic distance, in instructions, between a producer and a consumer. The y-axis shows the total number of consumers for each producer plotted. For instance, assume one of the consumer of a producer with fanout 5, is fetched 10 instructions apart from the producer. This will add one to the density (z value) of the point at coordinate position (x = 10, y = 5) in the graph.

We partition the graph into three regions, assuming a 32-entry instruction window: the region with five or more consumers (region B), the region with four or fewer consumers (region A), and the region outside of the instruction window (region C). The graph shows that the number of points in regions A and B are significant. It also shows, assuming that the ideal crossover point is in fact five consumers, that most instruction communication can happen within the window, and that the potential energy savings are large if the best communication mechanism can be selected for each pattern of operands.

Choosing the right communication mechanism based on the dependence pattern of each producing instruction and its consumers not only can effectively improve power efficiency but also can improve performance of the system. To achieve this goal, a system needs to leverage the producer-consumer knowledge available at runtime. In an ideal machine every producer knows exactly which consumers are waiting for its output right at the time it produces the output. In such a system, an efficient multicast mechanism that lets each producer directly wake up all its current consumers, may suffice regardless of operands pattern of the producer. In a real system, however, accurately providing this information for each producer can be challenging due to micro architectural or ISA restrictions. In the next subsection, we discuss these restrictions in the state-of-the-art architectures and the popular communication mechanisms used by those architectures.

### 2.2 Operand Bypassing Mechanisms

#### 2.2.1 Broadcast Bypass Networks

Communicating operands between instructions is a major source of energy consumption in superscalar processors [16]. Instruction dependences between producer and consumer instructions are dynamically extracted by the hardware. During register renaming phase, every instruction is assigned a tag (physical register) associated with its output. Later, when a consuming instruction for that first instruction is dispatched, the same tag will be assigned to the corresponding operand(s) of the consumer instruction. When the producer instruction is ready, its tag is broadcast to power-hungry CAMs, and all the instructions in the instruction queue compare the tags of their operands against the broadcast tag. If the broadcast tag and the tag of one of their operands are identical, the value of that operand will be set to the value read from the broadcast network or
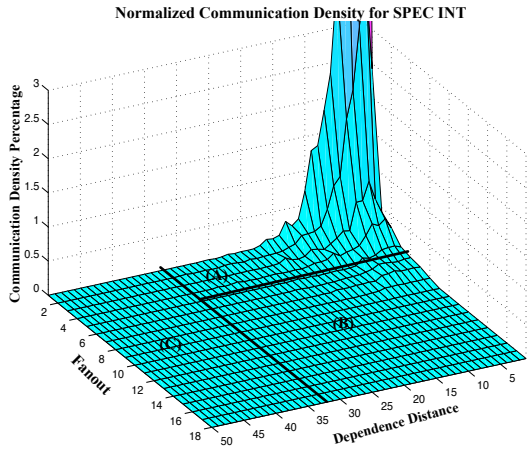
**Figure 1: Instruction communication distribution with different fanout and different dependence distances.**

the register file [16], and the ready flag of that operand is set.

Broadcast bypass networks are only power-efficient for instructions with high number of consumers in the instruction window. Figure 1 shows that instructions with this dependence pattern, which are represented in region B in that figure, constitute a small portion of all instructions in the window. Previous work [9] on SPEC benchmark shows that short dependences (region A in Figure 1) handled by the broadcast network constitute approximately 75% of program dependences in a superscalar processor. The cause for this significant inefficiency in superscalar processors is the way the dependence tree structure is built dynamically. This structure does not directly link producers to their current consumers in the window. Instead, consumer operands point to their producers using the physical tags. An alternative would be to change this structure such that each producer points directly to its consumers. In such an approach, when the the producer executes, it would only wake up its consumers, thus eliminating unnecessary tag matchings imposed by the bypass network approach. To achieve this goal, when a consumer instruction is dispatched, it updates a pointer set (or some other structure) associated with its producer instruction so that when the producer finishes executing, the producer has full or partial knowledge about its current consumers in the window. Based on this idea, there have been several recent studies [5, 6, 12, 18, 19] proposing such dynamic, hybrid communication models. For example, Gonzalez et al. [6] exploit the well-known phenomena that many instructions have small numbers of consumer instructions in the instruction window. Based on this observation, they propose a power-efficient issue logic design for superscalar processors. The approach employs an *N-use table*, which is indexed by physical registers, to store the first N consumer instructions of each physical register. If a physical register has more than N consumers in the table, the next consumer instruction is put into a small out-of-order instruction queue called I-buffer, to which a broadcast is performed. The instructions stored in the N-use table will acquire the operand through point-to-point communication, when the corresponding physical register is available, and the ones in the I-buffer will acquire the operand through a broadcast. The ratio between point-to-point and broadcast operations can be adjusted by changing the value of N. This approach eliminates most of the broadcasts and tag matchings and achieves high energy sav-

ings. On the other hand, each consumers may wait on multiple producers, multiple copies of consumer instructions can exist in the N-use table. These copies of an instruction in the N-use table need to maintain circular pointers to each other. These pointers need to be updated when the corresponding physical register is available. Consequently, the N-use table may require multiple read and write ports. Nonetheless, this approach is an effective way to increase efficiency.

Huang et al. [12] propose a full hardware pointer-based approach to eliminate the broadcasts and the tag matchings, which detects the one-consumer instructions dynamically and performs point-to-point communication to them. Any instruction targeting more than one instruction has to broadcast its result. During dispatch, a consumer instruction updates a pointer to itself in the instruction queue entry associated with its producer instruction. This pointer value is used during the issue of the producer instruction to directly send the result to the consumer. This approach does not use structures like N-use table. Instead, this approach employs point-to-point communication only for instructions that have only one consumer. When misprediction causes some instructions to flush, the pointers in the instruction queue pointing to the flushed instructions now longer are valid. As a result, this method requires a mechanism for cleaning up during branch misprediction.

These hybrid broadcast/point-to-point communication mechanisms significantly reduce consumed operand communication energy compared to superscalar models. In addition, they do not require any compiler or ISA support, which makes them flexible. However, these approaches support dynamically perform some pointer chasing and bookkeeping operations on structures like N-use table or instruction queus.

This paper evaluates a different hybrid approach in which with some ISA support, the compiler selects a mix of broadcast and point-to-point (token) bypasses between instructions in each block (a group of instructions) of the program. By moving the dependence pattern detecting task to the compiler, this approach does not hardware-based structures or pointer chasing operation. In addition, given the global knowledge about operand dependences and criticality of all instructions in each block at compile time, the compiler is able to effectively selecting the right communication mechanism for instructions. On the other hand, a drawback of using the compiler-assisted hybrid method is extra bits consumed in the ISA.

### 2.2.2 Tokens

Tokens or packets are used by dataflow machines for point-to-point communication among instructions. In this mechanism, each instruction encodes directly it destination instruction(s) through the ISA support. Dennis' dataflow machine [8] has an instruction memory with each instruction cell corresponding to an operation of a dataflow program. When the operands are ready, the instruction is sent through a high bandwidth switch to an operation unit to execute. After instruction is executed in the operation unit, the result of the operation is sent as one or two packets (or tokens), along with the address of a the destination operand to the instruction memory.

In first generation dataflow machines [8, 3], different from the conventional von Neumann machines, data values are not permanently stored in memory or registers. Instead, data values are transmitted among instructions using tokens allowing for massively parallel execution. However, these machines run programs written in specialized dataflow languages, which are not very popular. Another problem with using dataflow tokens, is variable instruction sizes that complicates the ISA design for such architectures.

This work relies on ISA support to combine static point-to-point (similar to dataflow tokens) and broadcast communication using
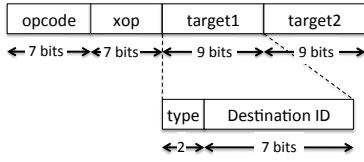
Figure 2: TFlex Instruction Encoding.

compiler analysis, which identifies the dependence pattern of instructions within each block of instructions. The compiler encodes static broadcast tags in the ISA only for critical high-fanout instructions within a block of instructions. These tags are used at runtime by the microarchitecture through a light-weight, energy-efficient broadcast mechanism. For the rest of the instructions in a block, the compiler encodes targets into the producers for direct point-to-point communication. By having the compiler choose between broadcasts and tokens depending the fanout of instructions, this mechanism achieves high power saving and performance improvements.

## 3. SYSTEM OVERVIEW

TFlex processor [13] implements an Explicit Data Graph Execution (EDGE) ISA [4], which supports block-atomic execution, and instructions within a block execute in dataflow order using tokens. Each block is allocated to one core [20] and is fetched into the instruction queue of that core. Different cores in the TFlex processor communicate through the memory and registers and a lightweight operand network. In each core, however, instructions directly communicate with their consumers within the instruction window. The original TFlex model supports different block mapping strategies [20]. In deep mapping which is used as our baseline in this work, each block is mapped to only one core. As a results, all intra-block communication is localized to the core.

The TFlex compiler [23] breaks the program into single-entry, predicated blocks of instructions, similar to hyperblocks [15]. In this ISA, each instruction encodes up to two target instructions in the same block using their offsets from the beginning of the block. If an instruction has more than two targets, the TFlex compiler [23] uses *move* instructions to generate a software fanout tree to deliver the output to its targets. Although this approach fixes the high-fanout instruction encoding problem, the inserted *move* instructions incur performance penalty in terms execution latency and the code size [10].

In this dataflow representation, each instruction explicitly encodes its target instructions in the same block using the offsets of the target instructions from the beginning of the block. For each instruction, its offset from the beginning of its block is the instruction ID of that instruction. An example of the initial intermediate code and its converted dataflow representation are shown in Figures 3(a) and 3(b), respectively. Instruction $i$ adds values $a$ and $b$ and sends the output to $operand_1$ and $operand_2$ of instructions $j$ and $k$, respectively. Instruction $j$ subtracts that value from another value $d$, and sends the output to $operand_2$ of instruction $k$. Finally, instruction $k$ stores the value computed by instruction $i$ at the address computed by instruction $j$.

Figure 2 illustrates instruction encoding used by the EDGE ISA. Because the maximum block size is 128 instructions, each instruction ID in the target field of a instruction requires seven bits. The target field also requires two bits to encode the type of the target because each instruction can have three possible inputs including $operand_1$, $operand_2$ and $predicate$.



9(a) Initial representation    9(b) Dataflow representation



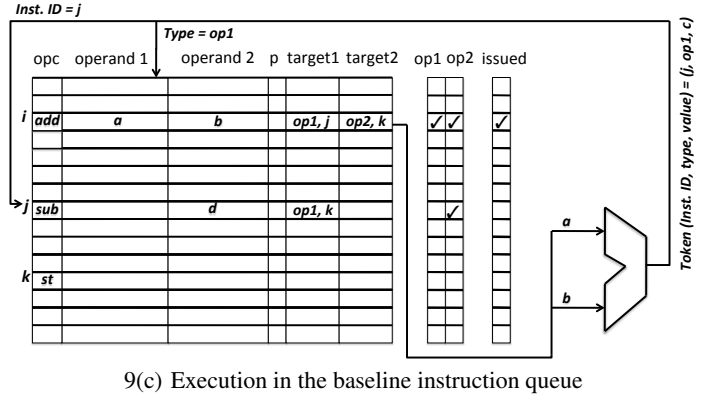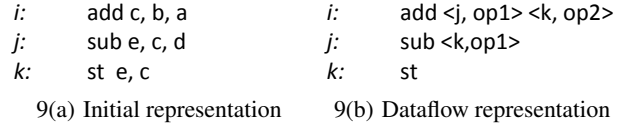9(c) Execution in the baseline instruction queue

Figure 3: A baseline code example.

Figure 3(c) illustrates a block diagram of the instruction queue of the baseline TFlex core when running the code sample. When instruction $i$ is issued, values $a$ and $b$ along with target of that instruction are brought to an execution unit. After instruction $i$ finishes executing, the execution unit sends two tokens to its targets one at a time. The first token is sent to the $operand_1$ of instruction $j$. This token, as shown in the figure, includes the instruction ID of the target instruction (value $j$ in the example), the target operand type ($operand_1$ in this example) and the output value of the executed instruction ($a + b$ in this example). The instruction ID and type fields of the token are used to directly index the rows and columns of the instruction queue and no tag matching is needed for finding the targets.

Although this instruction queue is very power-efficient as in other dataflow machines, it is not very performance-efficient when running high-fanout instructions. In the TFlex microarchitecture, one token is produced and bypassed for each target. TFlex uses *mov* instructions to generate fanout trees to distributed the output of high-fanout instructions. In the next section, we explain how we can exploit compiler-time producer-consumer information to augment this point-to-point communication model with a low-weight power-efficient broadcast mechanism.

## 4. HYBRID OPERAND COMMUNICATION MECHANISM

In this section proposes an approach for hybrid operand communication with compiler assistance. The goal of the new approach is to achieve higher performance and energy efficiency by allowing the compiler to choose best communication mechanism for each instruction during the compilation phase. The section discusses the implementation of the new approach, which consists of four parts: (1) heuristics to decide the operand communication mechanism during compilation; (2) ISA support for encoding the compiler decision, broadcast tags or point-to-point tokens; and (3) microarchitectural support for the hybrid communication mechanism. This section concludes with a discussion of design parameters and power trade-offs and performance implications of the proposed approach; (4) further compiler optimization to reuse broadcast tags

for instructions with non-overlapping broadcast live ranges.

## 4.1 Overview

Since each block of code is mapped to one core, the hybrid mechanism explained in this section is used to optimize the communication between instructions running within each core. This means that no point-to-point or broadcast operand crosses core boundaries. For cross-core (i.e. cross-block) communication, TFlex uses registers and memory [20], which are beyond the scope of this article. Of course extending hybrid communication to cross-core communication is an interesting area and can be considered future work of this work.

Different from dynamic hybrid models, the compiler-assisted hybrid model relies on the ISA to convey information about point-to-point and broadcast instructions into the microarchtecture. The involvement of the ISA leads provides some opportunities for the compiler while causing some challenges at the same time. Assuming a fixed instruction size, using tokens can lead to construction of fanout *move* trees and manifests itself at runtime in form of extra power consumption and execution delay. On the other hand, categorizing many instructions as broadcast instructions requires the hardware to use a wide CAM in the broadcast bypass network, which can become a major energy bottleneck. The main role of the compiler is to pick the right mixture of the tokens and broadcast such that the total energy consumed by the *move* trees and the broadcast network becomes as small as possible. In addition, this mixture should guarantee an operand delivery delay close to the one achieved using the fastest operand delivery method (i.e. the broadcast network). One challenge, however, is to find enough number of unused bits in the ISA to encode broadcast data and convey it to the microarchitecture.

## 4.2 Broadcast Tag Assignment and Instruction Encoding

One primary step in designing the hybrid communication model is to find a method to distinguish between low- and high-fanout instructions. instructions that can be analyzed. In the compiler-assisted hybrid communication approach, the compiler detects the high-fanout instructions and encodes information about their targets via the ISA. In this subsection, we first give an overview of the phases of the TFlex compiler. Then we explain the algorithm for detecting high-fanout instructions and the encoding information inserted by the compiled in the broadcast sender and receiver instructions.

The original TFlex compiler [23] generates blocks containing instructions in dataflow format by combining basic blocks using if-conversion, predication, unrolling, tail duplication, and head duplication. In each block, all control dependencies are converted to data dependencies using predicate instructions. As a result, all intra-block dependencies are data dependencies, and each instruction directly specifies its consumers using a 7-bit instruction identifier. As shown in Figure 2, each instruction can encode up to two target instructions in the same block. During block formation, the compiler identifies and marks the instructions that have more than two targets. Later, the compiler adds *move* fanout trees for those high-fanout instructions during the code generation phase.

The modified compiler for the hybrid model needs to accomplish two additional tasks, selecting the instructions to perform the broadcast, and assigning static broadcast tags to the selected instructions. The compiler lists all instructions with more than one target and sorts them based on the number of targets. Starting from the beginning of the list, the compilers assigns each instruction in the list a tag called broadcast identifier (BCID) out of a fixed number
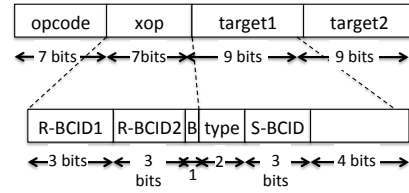


**Figure 4: TRIPS Instruction Encoding with Broadcast Support. S-BCID, R-BCID and B represents send BCID, receive BCID and the broadcast enable flag.**

| | |
|---|---|
| $i_1$: | add c, a, b |
| $i_2$: | sub e, c, d |
| $i_3$: | add f, c, g |
| $i_4$: | st d, c |
| $i_5$: | st f, e |

| | |
|---|---|
| $i_1$: | add $<i_2,\ op1> <i_{1a},\ op1>$ |
| $i_{1a}$: | mov $<i_3,\ op1> <i_4\ op1>$ |
| $i_2$: | sub $<i_5,\ op2>$ |
| $i_3$: | add $<i_5,\ op1>$ |
| $i_4$: | st |
| $i_5$: | st |

9(a) Initial representation     9(b) Dataflow representation

| | |
|---|---|
| $i_1$: | add [SBCID=1, op1] |
| $i_2$: | sub [RBCID=1] $<i_5,\ op1>$ |
| $i_3$: | add [RBCID=1] $<i_5,\ op1>$ |
| $i_4$: | st [RBCID=1] |
| $i_5$: | st |

9(c) Hybrid dataflow/broadcast representation

**Figure 5: A sample code and corresponding code conversions in the modified compiler for the hybrid model.**

ber of BCIDs. For producers and consumers send or receive BCIDs needs to be encoded inside each instruction. Therefore, the total number of available BCIDs is restricted by the number of unused bits available in the ISA. Assuming there are at most $MaxBCID$ BCIDs available, then the first $MaxBCID$ high-fanout instructions in the list are assigned a BCID.

After the broadcast sender instructions are detected and BCIDs are assigned, the compiler encodes the broadcast information inside the sender and receiver instructions. Figure 4 illustrates the ISA extension using a sample encoding for $MaxBCID$ equal to eight. Each sender contains a broadcast bit, bit $B$ in the figure, enabling broadcast send for that instruction. The compiler also encodes the BCID of each sender inside both the sender and the receiver instructions of that sender. For the sender, the target bits are replaced by the three send BCID bits and two broadcast type bits. Each receiver can encode up to two BCIDs with six bits, and so it can receive its operands from two possible senders. Although this encoding uses two BCIDs for each receiver instruction, the statistics show that a very small percentage of instructions may receive broadcasts from two senders. For the other instructions that are not receiver of any broadcast instructions, the compiler assigns the receive BCIDs to 0, which disables the broadcast receiving mechanism for those instructions.

Figure 5 illustrates a sample program (except for *stores*, the first operand of each instruction is the destination), its equivalent dataflow representation, and its equivalent hybrid token/broadcast representation generated by the modified compiler. In the original dataflow shown code 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 com-

piler assigns a BCID (BCID of 1 in this example) to $i_1$, the instruction with high fanout, and eliminates the *move* instruction. The compiler also encodes the broadcast information into the $i_1$ and its consuming instructions (instructions $i_2$, $i_3$ and $i_4$). The compiler use tokens for the remaining low-fanout instructions. For example, instruction $i_3$ has only one target (instruction $i_5$) so $i_3$ still uses token-based communication. In the next subsection, we explain how these fields are used during the instruction execution and what additional optimizations are possible in the proposed hardware implementation.



**Figure 6: Execution of a broadcast instruction in the IQ.**



**Figure 7: Compare logic of BC CAM entries.**

### 4.3 Microarchitectural Support

To implement the broadcast communication mechanism in the TFlex substrate, a small CAM array is used to store the receive BCIDs of broadcast receiver instructions in the instruction queue. When instructions are fetched, the receive BCIDs are stored in a CAM array called $BCCAM$. Figure 6 illustrates the instruction queue of a single TFlex core when running the broadcast instruction $i_1$ in the sample code shown in Figure 5(c). When the broadcast instruction executes the broadcast signal, bit $B$ in Figure 4 is detected, then the sender BCID (value *001* in this example) is sent to be compared against all the potential broadcast receiver instructions. Notice that only a subset of instructions in the instruction queue are broadcast receivers and the rest of them need no BCID comparison. Among all receiving instructions, the tag comparison will match only for the CAM entries corresponding to the receivers of the current broadcast sender (instructions $i_2$, $i_3$ and $i_4$ in this example). Each matching entry of the $BCCAM$ will generate a write-enable signal to enable a write to the operand of the corresponding receiver instruction in the RAM-based instruction queue. The broadcast type field of the sender instruction ($operand1$ in this example) is used to select the column corresponding to the receivers' operand, and finally all the receiver operands of the selected type are written simultaneously into the instruction window.

It is worth noting that tag delivery and operand delivery do not happen at the same cycle. Similar to superscalar operand delivery networks, the tag of the executing sender instruction is first delivered at the right time, which is one cycle before instruction execution completes. At the next cycle, when instruction result is ready, the result of the instruction is written simultaneously into all waiting operands in the instruction window.

Figure 7 illustrates a sample circuit implementation for the compare logic in each $BCCAM$ entry. The CAM tag size is three bits which represents a $MaxBCID$ parameter of eight. In this circuit, the compare logic is disabled if one of the following conditions is true:

- If the instruction corresponding to the CAM entry has been previously issued.

- If the receiver BCID 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 the example shown in Figures 6 and 5.

- If the executed instruction is not a broadcast sender.

This hybrid broadcast model is more energy-efficient than the instruction communication model in superscalar processors for several reasons. First, because of the $MaxBCID$ limit on the maximum number of broadcast senders, the size of the broadcast tag, which equals to the width of the CAM, could be reduced from $Log(InstructionQueueSize)$ to $Log(MaxBCID)$. A broadcast consumes significantly less energy because it drives a much narrower CAM structure. Second, only a small portion of bypasses
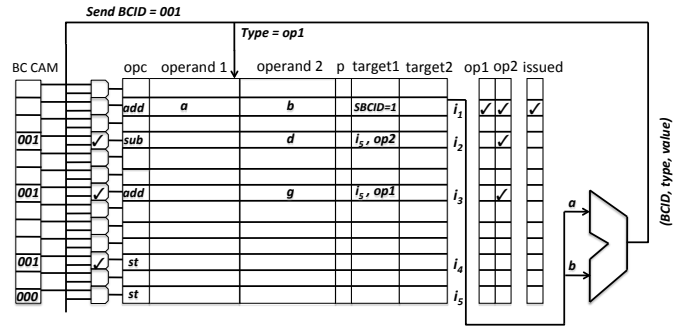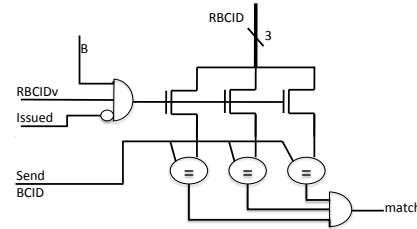
are selected to be broadcast and the majority of them use the token mechanism, since the compiler only selects a portion of instructions to perform broadcasts. Third, only a portion of instructions in the instruction queue are broadcast receivers and perform BCID comparison during each broadcast. Both of these design aspects are controlled by the $MaxBCID$ parameter. This parameter directly controls the total number of broadcast senders in the block. On the other hand, as we increase the $MaxBCID$ parameter, the number of active broadcast targets is likely to increase, but the average number of broadcast targets per broadcast is likely to shrink.

Different values of $MaxBCID$ represent different design points in a hybrid broadcast/token communication mechanism. $MaxBCID$ of zero represents a pure token-based communication mechanism and fanout trees using *move* instructions. $MaxBCID$ of 128 means every instruction with fanout larger than one will be a broadcast sender. In other words, the compiler does not analyze any global fanout distribution to select right communication mechanism for each instruction. Instead, all fanout instruction in each block use broadcast operation. This model is close to a TFlex implementation of a dynamic hybrid point-to-point/broadcast communication model [12]. It is worth mentioning that even with $MaxBCID$ equal to 128, there are still many instructions with just one target and those instructions still use token-based communication. As we vary the $MaxBCID$ form zero to 128, more fanout trees are eliminated, and more broadcasts are added to the system. By choosing an appropriate value for this parameter, the compiler is able to minimize total power consumed by fanout trees and broadcasts while achieving a decent speedup in performance as a result of using broadcasts for high-fanout instructions.

### 4.4 Further Compiler Optimization: Reusing Broadcast-ID by ID-Allocation

This section proposes a compiler optimization that reuses broadcast IDs in each instruction block. The goal of this optimization is to support more architectural broadcasts than $MaxBCID$ provided by $BCCAM$, thus achieving higher performance while spend-
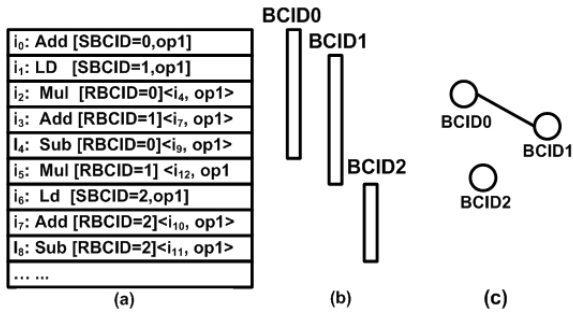
**Figure 8: An example shows the potential of reusing broadcast-ID. (a)Code layout in instruction queue. (b)the necessary live-range of each broadcast-ID. (c) the interference-graph of the broadcast-ID live-range**
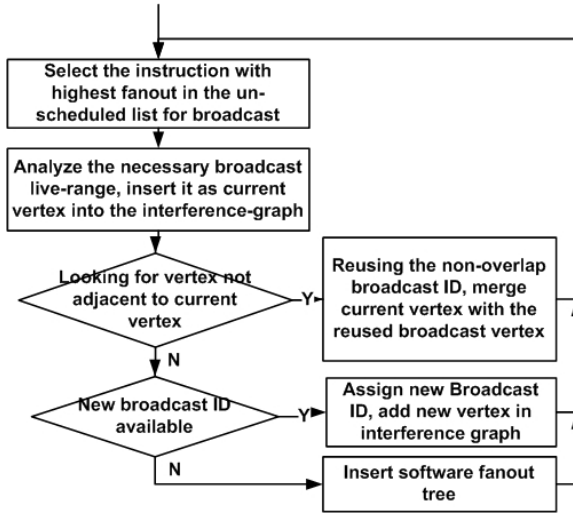


**Figure 9: Flowchart of the Broadcast ID Allocation Algorithm**



**Figure 10: Single Entry Design of the new CAM. Note: 8bit one-hot encoding is used**

ing lower energy.

In the basic compiler-assisted approach, as discussed in previous section, each broadcast ID is only assigned to one producer-instruction which broadcasts this ID to all entries in the CAM while most of the waiting instructions are not its consumer. In this model, the CAM width limits the number of broadcast instructions because the number of available architectural broadcasts depends on the width of the CAM.

The idea of reusing broadcast-ID by ID allocation is based on the observation that if two broadcast instructions are not sharing the range from its first consumer-instruction to the last one in the instruction queue, the two broadcasts could use the same broadcast ID, as long as each broadcast does not send its result to the consumers of another broadcast with the same broadcast ID .

Figure 8 gives an example to show the potential for reusing broadcast ID. The codes are in the instruction queue. There are three broadcast instructions ($i0$, $i1$, $i6$), each of which has been assigned a broadcast ID shown as SBCID. The consumer instructions of a broadcast ID have the same RBCID which all equal to sender's SBCID. There is no broadcast range overlap between $i0$ and $i6$. If the microarchitecture can guarantee that $i0$ can only broadcast to the instruction in the region below $i0$ and above $i6$, $i0$ and $i6$ can share the same broadcast ID.
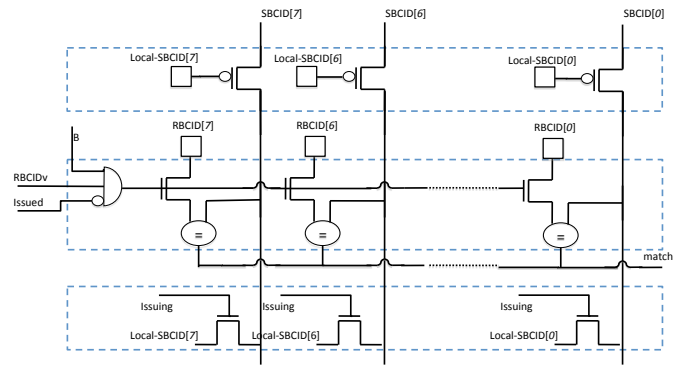
### 4.4.1 Broadcast-ID Allocation in the Compiler

Detecting and assigning reusable broadcast IDs are similar to the register allocation problem. We define the necessary live-range of each broadcast ID as smallest range of instructions in the instruction queue to which that the broadcast ID need to be sent. Clearly, this live-range ends at the last consumer instruction of that broadcast. The necessary live-ranges of the three broadcast IDs are shown in Figure 8(b). Based on the necessary live-range of each broadcast, the interference-graph can be built representing each broadcast ID with a vertex and overlaps between . In such graph, if the necessary live-ranges of two broadcast instructions overlap, the two corresponding vertexes are connected through an edge. The interference-graph of the example code is showed in Figure 8(c). The compiler can safely assign reusable broadcast IDs by checking non-adjacent vertexes in the interference graph. Furthermore, the compiler can schedule the instructions to minimize the overlap among necessary broadcast live-ranges.

A simple algorithm has been proposed and evaluated based on the flowchart shown in Figure 9. Algorithm optimizations will be part of the future work.

### 4.4.2 Microarchitectural Support

Reusing broadcast ID requires the microarchitectural support to guarantee that each broadcast will be sent to instructions below it up to the next broadcast instruction with same broadcast ID. This section proposes the CAM hardware design that satisfies the requirement with minimum energy consumption.

The new CAM design has mainly two new features: (1) Each CAM entry stores the decoded one-hot code of broadcast ID, the decoding happens when instruction get fetched. Each broadcast producer will only drive one wire on the broadcast-ID bus which we call it the hot-wire of this entry. (2) Each CAM entry disconnects its own hot-wire above itself. Each wire in the broadcast ID bus is functionally divided into several segment so that the instructions sharing the same broadcast ID will each drive its own segment of same wire. Figure 10 shows the design a single entry of the new CAM. In order to support eight architectural broadcasts, the one-hot encode broadcast-ID needs an 8-bit bus (*BCID[7:0]* in the figure). Each entry consists of three parts: the gating logic, the comparing logic and the driving logic. Compared to the original designed, the only added hardware is the gating logic, which disconnects the hot-wire of the entry above this entry. Comparing with normal CAM design, this new CAM costs more hardware, however, each broadcast will only cause gate switch on a segment of a single hot-wire. Thus, each broadcast will consume less energy.

# 5. EVALUATION AND RESULTS

In this section we evaluate the energy consumption and performance of the compiler-assisted hybrid operand communication model. We first describe the experimental methodology followed by statistics about the distribution of broadcast producers and consumers. This distribution data will indicate the fraction of all instructions in the window that have a high fan-out value. The distribution also suggests the minimum $MaxBCID$ and $BCCAM$ bit-width needed for assigning broadcast tags to all of those high-fanout instructions. Then, we report performance results and power breakdown of fanout trees or broadcast instructions for different $MaxBCID$ values. After that, the dynamic hybrid approach is compared with this compiler-assisted approach. Finally the broadcast-ID reusing mechanism is evaluated. These results show that by intelligently picking a subset of high-fan out instructions for broadcast, the compiler is able to reduce the total power significantly without losing much performance than if it picked all high-fanout instructions.

The results show that this compiler-assisted hybrid model consumes significantly lower power than the pure broadcast mechanism used by superscalar processors. With this hybrid communication model, we explore the full design space ranging from a very power efficient token-based dataflow communication model to a high-performance broadcast model similar to that used in superscalar machines. The results show that the compiler assistance is more reliable than dynamically choosing the right operand communication mechanism for each instruction. Given the compiler assistance, not only are we able to achieve a higher energy efficiency than pure dataflow, but at the same time we are also able to achieve better performance in this design space.

## 5.1 Methodology

We augment the TFlex simulator [13] with the support for the hybrid communication model explained in the previous section. In addition we modify the TFlex compiler to detect high-fanout instructions and to encode broadcast identifiers in those instructions and their targets. Each TFlex cores is a dual-issue, out-of-order core with a 128-instruction window. Table 2 shows the microarchitectural parameters of each TFlex core. The energy consumed by move instructions during the dispatch and issue phases is already incorporated into original TFlex power models [13]. We augment the baseline TFlex models with the power consumed in the $BCCAM$ entries, modeled using CACTI 4.1 [7], when tag comparisons are made during a broadcast.

The results presented in this section are generated using runs on several SPEC INT [2] and EEMBC [1] benchmarks running on 16 TFlex cores. We use seven integer SPEC benchmarks with the reference (large) dataset simulated with single SimPoints [22]. The SPEC FP benchmarks achieve very high performance when running on TFlex, so the speedups are less important and interesting to this work. We also use 28 EEMBC benchmarks which are small kernels with various characteristics. We test each benchmark varying the $MaxBCID$ from 0 to 128 to measure the effect of that parameter on different aspects of the design.

## 5.2 Distribution of Producers and Operands

Figure 11 shows the average cumulative distribution of the number of producers and the operands for different fanout values for SPEC INT benchmarks. The cumulative distribution of producers converges much faster that the one of operands does, which indicates a small percentage of producers corresponds to a large number of operands. For example, for fanouts larger than four, only 8% of producers produce 40% of all operands. It indicates that performing broadcasts on a small amount of producers could improve
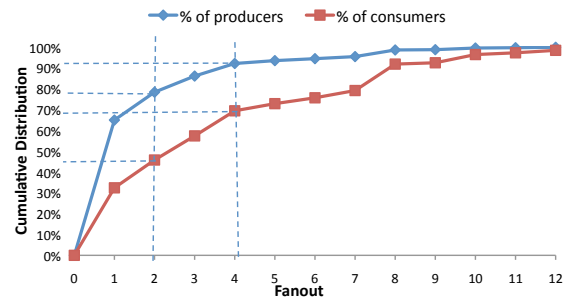


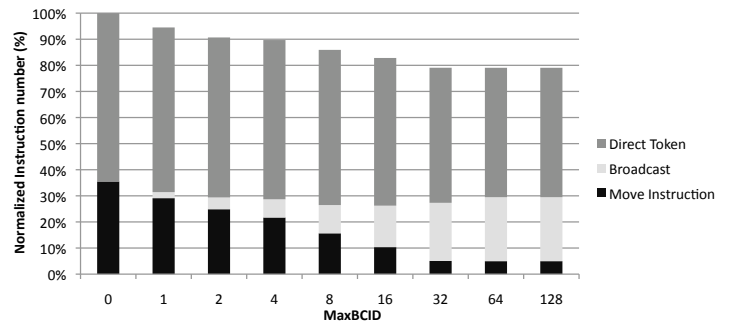**Figure 11: Cumulative distribution of producers and operands.**



**Figure 12: The ratio of broadcast, move and other instructions.**

operand delivery for a large number of operands. The information shown in this graph is largely independent from the microarchitecture and reflects the operand communication behaviors of the programs. To choose the right mechanism for each producer, one also must consider the hardware implementation of each mechanism. This graph shows that 78% of all instructions have fanout equal or less than two. For these instructions, given the TFlex microarchitecture, it is preferred to use efficient token-based communication. For the rest of instructions, finding the right breakdown of instructions between broadcasts and *move* trees also depends on the cost of each of these mechanisms.

Figure 12 shows the breakdown ratio of broadcast producers, instructions sending direct tokens, and the *move* instructions to all instructions for the SPEC benchmarks when using the compiler-assisten model proposed in this paper. The number of broadcast instructions (producers) increases dramatically for smaller $MaxBCID$ values, but levels off as the $MaxBCIDs$ parameter approaches 32. At the same time, the ratio of move instructions decreases from 35% to 5%. As a result, the total number of instructions drops to 79%. This observation indicates that the compiler can detect most of the high-fanout dependences inside a block and replace the software fanout tree by using only up to 32 broadcasts. The data shown in Figure 12 also indicates that even with the unlimited number of broadcasts, at most 25% of the instructions use broadcast communication and the rest of them use tokens for communicating. This is almost one fourth of the number of broadcasts used by a superscalar machine because in a superscalar machine all instructions must use the broadcast mechanism. Another observation is that the total number of instructions decreases 15% with only 8 broadcasts, which indicates that a small number of broadcasts could give us most of the benefits of unlimited broadcasts.

## 5.3 Energy Tradeoff

**Table 2: Single Core TFlex Microarchitecture Parameters [13]**

| Parameter | Configuration |
|---|---|
| Instruction Supply | Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Num. entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256. |
| Execution | Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP) or single issue. |
| Data Supply | Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 44-entry LSQ bank; 4MB decoupled S-NUCA L2 cache [14] (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles. |
| Simulation | Execution-driven simulator validated to be within 7% of real system measurement |

Figure 13 illustrates the energy breakdown into executed *move* and broadcast instructions for a variety of $MaxBCID$ values on the SPEC benchmarks. The energy values are normalized to the total energy consumed by *move* instructions when instructions communicate only using tokens ($MaxBCID = 0$). When only using tokens, all energy overheads are caused by the *move* instructions. Allowing one or two broadcast instructions in each block, $MaxBCIDs$ of 1 and 2, we observe a sharp reduction in the energy consumed by *move* instructions. As discussed in the previous section, the compiler chooses the instructions with highest fanout first when assigning BCIDs. Consequently, high number of *move* instructions are removed for small $MaxBCIDs$ which results in significant reduction in the energy consumed by *move* instructions. For these $MaxBCIDs$ values, the energy consumed by broadcast instructions is very low.

As we increase the total number of broadcast instructions, the energy consumed by broadcast instructions increases dramatically and fewer move instructions are removed. As a result, at some point, the broadcast energy becomes dominant. For high numbers of $MaxBCID$, the broadcast energy is orders of magnitude larger than the energy consumed by *move* instructions. The key observation in this graph is that for $MaxBCID$ equal to 4 and 8, in which only 4 to 8 instruction broadcast in each block, the total energy consumed by *moves* and broadcast is minimum. For these $MaxBCIDs$, the total energy is about 28% lower than the energy consumed by a fully dataflow machine ($MaxBCID = 0$) and about 2.7x lower than when $MaxBCID$ is equal to 128. These results show that the compiler is able to achieve a better trade-off in terms of power breakdown by selecting a critical subset of high-fanout instructions in each block. We also note that for $MaxBCIDs$ larger than 32, the energy consumed by *move* instructions is at a minimum and does not change. In an ideal setup where the overhead of broadcast is ignored, these points give us the best possible energy savings. This energy is four time lower than the total energy consumed when using $MaxBCID$ equal to 8, which is the point with the lowest total power. The energy breakdown chart for EEMBC benchmarks is similar to SPEC benchmarks except that $MaxBCID$ of 4 results in lower total power consumption than $MaxBCID$ of 8.

environment. In order to choose the best communication mechanism for each instruction, the analytical model measures the energy consumption of a single *move* instruction and that of broadcast CAMs of different bit widths. The energy consumption of software fanout tree mainly comes from several operations, such as writing/reading move instructions in the instruction-queue, writing/reading operands in the operand buffer, generating control signals, driving the interconnection wires which includes the activities on the wire networks when fetching, decoding, executing of the move instruction and transmitting the operand. On the other side, the energy consumption of the broadcast operations mainly comes from driving the CAM structure, the tag-matching and writing the operands in the operand buffer. The energy consumed by each of these operations is modeled and evaluated with CACTI4.1 [7] and the power model in the TFlex simulator [13], and used by the analytical model. For a specific $MaxBCID$ $x$, the analytical model estimates the lower bound of energy consumption of the hybrid communication model assuming an ideal situation in which that there are unlimited number of broadcast tags and each broadcast consumes as little energy as a broadcast using a CAM width $logx$. Based on this assumption, the analytical model finds the break even point between *moves* and broadcast instructions in which the total energy consumed by broadcasts is the same as the total energy consumed by *moves*.

As can be seen in Figure 13, for small or large values of $MaxBCID$, the real total power consumed by *moves* and broadcasts is significantly more than the ideal energy estimated by the analytical model. This difference seems to be minimum when $MaxBCID$ equals 8, which the total consumed power is very close to the optimum power at this point. Table 3 reports the percentage of broadcast producer instructions for different BCIDs achieved using ideal analytical model and compiler-assisted approach. With small $MaxBCIDs$, the large difference between real energy and ideal energy is because there is not enough tags to encode more broadcasts. On the other hand, when using large $MaxBCIDs$ the more than enough number of broadcasts are encoded, which increases the energy consumption. Finally, with $MaxBCIS$ of eight, the percentage of broadcast is very close to that achieved using the ideal analytical model.

We also measured the total energy consumption of the while processor (including SDRAMs and L2 caches) with variable $MaxBCID$. The compiler-assisted hybrid communication model achieves 6% and 10% total energy saving for SPEC INT and EEMBC benchmarks, respectively. The energy reduction mainly comes from two aspects: (1) replacing software fanout trees with broadcasts which reduces the energy of instruction communication; (2)reducing the total number of instructions , so there are fewer number of I-Cache access (and misses) and less overhead for executing the move instructions.

| Max BCID | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|---|
| Compiler-assisted | | 2 | 5 | 8 | 13 | 19 | 28 | 31 | 31 |
| Ideal | | 35 | 35 | 35 | 14 | 14 | 14 | 6 | 6 |

**Table 3: Percentage of broadcast producers for real and ideal models.**

Figure 13 also shows the lower bound energy consumption values derived using an analytical model. This analytical model gives us the best communication mechanism for each producer in an ideal
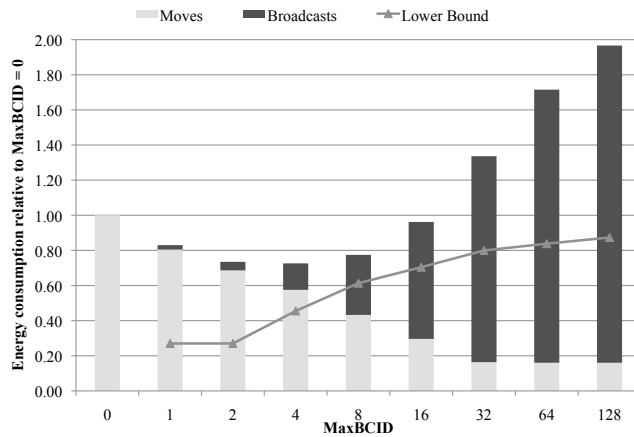
## 5.4 Performance Improvement

**Figure 13: Averaged energy breakdown between move instructions and broadcasts for various MaxBCIDs for SPEC benchmarks.**
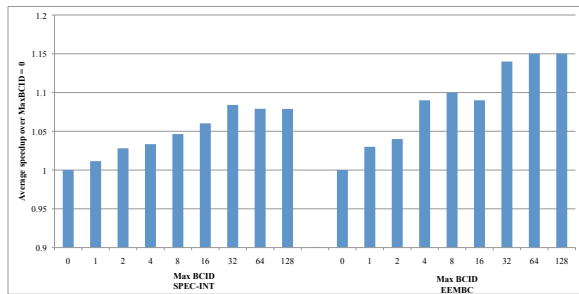


**Figure 14: Average speedups achieved using various MaxBCIDs for SPEC and EEMBC benchmarks.**

In terms of performance, full broadcast has the potential to achieve highest performance. The reasons are that there is only one cycle latency between the broadcast instructions with its consumers, while communicating the operands though move tree results in more than one cycle latency. However, large number of broadcast causes large amount of energy consumption. There is an important trade-off between the performance and the energy efficiency when using viable value of $MaxBCID$. This subsection evaluates the performance improvement for different parameters. The key observation from the evaluation is that 8 broadcasts per-block could be the best tradeoff between the performance and energy efficiency. It achieves most of the speedup reached by the unlimited broadcast, at the same time, it saves most of the energy as discussed in last subsection.

Figure 14 shows the average performance improvement over TFlex cores with no broadcast support ($MaxBCID$ = 0) for the SPEC and EEMBC benchmarks.

The average speedup reaches its maximum as $MaxBCID$ reaches 32 and remains almost unchanged for larger values. As shown in Figure 12, with $MaxBCID$ equal to 32, most of high-fanout instructions are encoded. The speedup achieved using $MaxBCID$ of 32 is about 8% for SPEC benchmarks. Again, for the EEMBC benchmarks $MaxBCID$ of 32 achieves very close to the best speedup, which is about 14%. On average, the EEMBC benchmarks gain higher speedup using the hybrid approach, which might be because of larger block sizes in EEMBC applications, which provide more opportunity for broadcast instructions. Most EEMBC
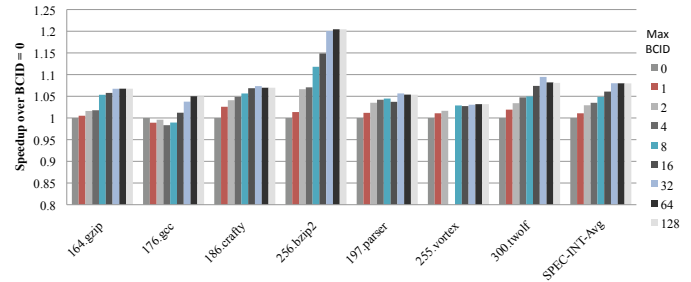


**Figure 15: Speedups achieved using various MaxBCIDs for individual SPEC benchmarks.**

benchmarks consist of parallel loops, whereas the SPEC benchmarks have a mixture of small function bodies and loops. In addition, the more complex control flow in SPEC benchmarks results in relatively smaller blocks.

Figure 15 shows the performance improvement over TFlex cores with no broadcast support ($MaxBCID$ = 0) for individual SPEC benchmarks. The general tend for most benchmarks is similar. We do not include the individual EEMBC benchmarks here because we notice similar trends in EEMBC too. For $gcc$, the trend of speedups is not similar to other benchmarks for some $MaxBCID$ values. We attribute this to the high misprediction rate in the memory dependence predictors used in the load/store queues.

Although $MaxBCID$ of 32 achieves the highest speedup, but Figure 13 shows it may not be the most power-efficient design point compared to the power-efficiency of full dataflow communication. When designing for power-efficiency, one can choose $MaxBCID$ of 8 to achieve the lowest total power, while still achieving a decent performance gain. Using $MaxBCID$ of 8 the speedup achieved is about 5% and 10% for SPEC and EEMBC benchmarks, respectively, and the power is reduced by 35%.

| Max BCID | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|---|
| Blocks | 1 | 0.95 | 0.94 | 0.92 | 0.9 | 0.88 | 0.84 | 0.84 | 0.84 |
| Inst.s | 1 | 0.95 | 0.90 | 0.87 | 0.83 | 0.79 | 0.76 | 0.76 | 0.76 |

**Table 4: The average number of executed blocks and instructions using various MaxBCIDs.**

As Table 4 shows there is a decrease in the number of blocks and instructions executed as we increase *MaxBCID*. This directly translates into improved performance because by eliminating *move* instructions we reduce resource contention in the ALUs and at the same time decrease the total number of tokens. The decrease in the number of blocks executed reduces pressure on the instruction cache and also reduces the total cost of control operations since each block incurs fixed control overheads.

## 5.5 Comparison with the Dynamic Hybrid Communication Approach

There have been several studies [5, 6, 12, 18, 19] proposing dynamic hybrid communication models, which rely on hardware to track and analyze the related information then decide the communication mechanism for each instruction. As analyzed in previous section, dynamic approaches have different tradeoff comparing with compiler-assisted approach. We compare the compiler-assisted approach with one of latest dynamic approaches [12], which
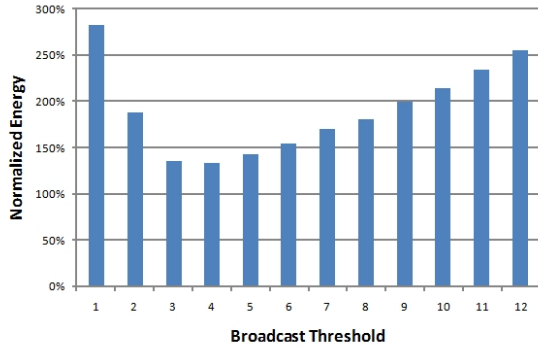
**Figure 16: Normalized energy consumption of selected dynamic hybrid approach with variable thresholds. Note: the energy values have been normalized to the energy of best configuration in the compiler-assisted approach without the ID-reusing optimization**



**Figure 17: Normalized dynamic move-instruction numbers with and without reusing ID**

broadcasts from instructions with fanout bigger than the predefined threshold(one), and performs point-to-point communication from others based on a pointer table attached with the register-alias-table. The authors in [12] make the observation that the pointer table becomes too complicated with more than one-pointer per-entry, so they do not exploit the designs with thresholds other than one. We extend their approach with variable thresholds in the TFlex processor toolchain to exploit whole design space, and estimate the power consumption with TFlex power model augmented with CACTI4.1 [7].

Figure 16 shows the energy consumption of the TFlex implementation of selected dynamic hybrid approach [12] with variable thresholds. The energy values have been normalized to the energy of best configuration in the compiler-assisted approach without reusing ID. When the threshold equals to one, the dynamic model consumes 2.8X instruction communication energy compared to the compiler-assisted model. When the threshold becomes bigger than one, it requires multiple pointers per-entry in the pointer table, which requires significantly complicated hardware as observed in [12]. When ignoring hardware complexity to only compare the energy consumption, as shown in the figure, the dynamic hybrid design with best configuration consumes 1.3X energy of that compiler-assisted approach does. The comparison validates that the static approach reaches higher energy efficiency by relying on compiler to select the communication mechanism.

## 5.6 Reusing Broadcast IDs

The compiler-assisted hybrid approach enables the system reuse the broadcast IDs, so that more broadcast instructions could be supported with fewer architectural broadcasts, which results in both higher performance and less energy. Figure 17 shows the normalized dynamic move instruction numbers for variable $MaxBCID$ with and without reusing broadcast ID. As shown in this figure, with reusing IDs, larger ratio of move instructions in the fanout tree could be removed while $MaxBCID$ ranges from 1 to 16. Reusing broadcast ID could enable almost 2X instructions to broadcast. Four architectural broadcasts with reusing ID results in similar amount of move instructions as the eight architectural broadcasts do without reusing . For $MaxBCID$ beyond 16, all the removable move instructions have been eliminated, so reusing broadcast-ID makes no difference.

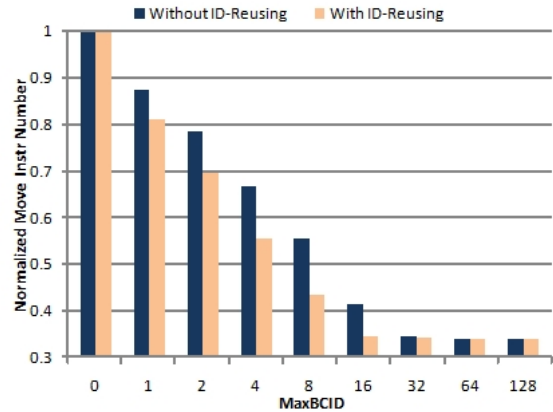In the energy consumption perspective, the new CAM design uses the one-hot encoding and each broadcast only drives a segment of a single hot-wire, which will cause less gate-switching. Thus less energy consumption could be expected.

## 6. CONCLUSIONS

Given the slowdown in supply voltage scaling, power and energy are now such huge concerns that anything architects can do to reduce them, without hampering performance, is worthwhile. Communication of operands between instructions, particularly in a clustered or distributed processor, is one of the major sources of energy consumption in high-end, out-of-order processors. This communication is therefore a good target for energy reduction. To reduce energy consumed during operand delivery in superscalar bypass network, several dynamic hybrid communication models have been proposed. These models dynamically trace producer-to-consumer information between instructions using hardware-based methods such as hardware pointing chasing operations in the instruction window. By having low fanout producer instructions wake up only their consumers, these models eliminate unnecessary tag matchings imposed by the bypass network approach.

This paper proposes a compiler-assisted hybrid operand communication model. Instead of using dynamic hardware-based pointer chasing, this method relies on the compiler to categorize instructions for token or broadcast operations. In this model, the compiler took a simple approach: broadcasts were used for operands that had many consumers, and dataflow tokens were used for operands that had few consumers. The compiler can analyze the program in a bigger range to select the best operand communication mechanism for each instruction. At the same time, the block-atomic EDGE model made it simple to perform that analysis in the compiler, and allocate a number of architecturally exposed broadcasts to each instruction block. Furthermore, compiler performs complex optimizations without hardware cost and execution-time penalty like the dynamic approaches do. The proposed broadcast-ID-reusing mechanism is a good example for complex optimizations better to be performed with compiler. By limiting the number of broadcasts, the CAMs searching for broadcast IDs can be kept narrow, and only those instructions that have not yet issued and that actually need a broadcast operand need to be performing CAM matches. This approach is quite effective at reducing energy; with eight broadcast IDs per block, 28% of the instruction communication energy is eliminated by eliminating many move instructions (approximately

55% of them), and performance is improved by 8% on average due to lower issue contention, reduced critical path height, and fewer total blocks executed. In addition, the results show that the power savings achieved using this model are close to the minimum possible power savings using a near-ideal operand delivery model.

The major downside to this approach is the encoding space required for specifying receivers of broadcast operands. Because EDGE ISAs encode destination consumers in each instruction, rather than source operands, encoding a sender's broadcast fits nicely within the existing TFlex instruction formats. Encoding the consumer's broadcast ID, however, does not line up well with the ISA model, and requires a number of bits equal to the logarithm of the number of broadcasts.

## 7. REFERENCES

[1] The embedded microprocessor benchmark consortium (EEMBC), http://www.eembc.org/.

[2] The standard performance evaluation corporation (SPEC), http://www.spec.org/.

[3] K. Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Computer*, 39(3):300–318, 1990.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and others. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, pages 44–55, July 2004.

[5] Ramon Canal and Antonio González. A Low-complexity Issue Logic. In *Proceedings of the 14th International Conference on Supercomputing*, pages 327–335, 2000.

[6] Ramon Canal and Antonio González. Reducing the Complexity of the Issue Logic. In *Proceedings of the 15th International Conference on Supercomputing*, pages 312–320. ACM, 2001.

[7] S. Thoziyoor D. Tarjan and N. Jouppi. HPL-2006-86, HP Laboratories, technical report. 2006.

[8] Jack B. Dennis and David P. Misunas. A Preliminary Architecture for a Basic Data-flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer architecture*, pages 126–132, 1975.

[9] Manoj Franklin and Gurindar S. Sohi. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-grain Parallel Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, 1992.

[10] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An Evaluation of the TRIPS Computer System. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2009.

[11] Dan Gibson and David A. Wood. Forwardflow: A Scalable RAM-Based Core. In *University of Wisconsin Computer Sciences Technical Report #1656*, 2009.

[12] Michael Huang, Jose Renau, and Josep Torrellas. Energy-efficient Hybrid Wakeup Logic. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 196–201, 2002.

[13] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–394, Chicago, Illinois, USA, 2007. IEEE Computer Society.

[14] Chankyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[15] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *25th Annual International Symposium on Microarchitecture*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[16] Dmitry V. Ponomarev, Gurhan Kucuk, Oguz Ergin, Kanad Ghose, and Peter M. Kogge. Energy-efficient Issue Queue Design. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(5):789–800, 2003.

[17] S.W. Keckler R. Desikan, D.C. Burger and T.M. Austin. Sim-alpha: A Validated, Execution-Driven Alpha 21264 Simulator. *UT-Austin Computer Sciences Technical Report TR-01-23*, October 2001.

[18] Marco A. Ramirez, Adrian Cristal, Mateo Valero, Alexander V. Veidenbaum, and Luis Villa. A New Pointer-based Instruction Queue Design and Its Power-Performance Evaluation. In *Proceedings of the 2005 International Conference on Computer Design*, pages 647–653, 2005.

[19] Marco A. Ramírez, Adrian Cristal, Alexander V. Veidenbaum, Luis Villa, and Mateo Valero. Direct Instruction Wakeup for Out-of-Order Processors. In *IWIA '04: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 2–9, 2004.

[20] Behnam Robatmili, Katherine E. Coons, Doug Burger, and Kathryn S. McKinley. Strategies for Mapping Dataflow Blocks to Distributed Hardware. In *International Conference on Microarchitectures*, pages 23–34, 2008.

[21] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *30th Annual International Symposium on Computer Architecture*, pages 422–433, june 2003.

[22] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[23] 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*, March 2006.

[24] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. WaveScalar. In *36th Symposium on Microarchitecture*, December 2003.