Hybrid Operand Communication for Dataflow Processors

Dong Li Behnam Robatmili Sibi Govindan Doug Burger Steve Keckler
University of Texas at Austin, Computer Science Department
{dongli, beroy, sibi, dburger, skeckler}@cs.utexas.edu

Abstract

One way to exploit more ILP and improve the performance of a single threaded application is to speculatively execute multiple code regions on multiple cores. In such a system, 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. lThis 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. 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. 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

Improvement of the single thread performance highly relies on the amount of ILP that can be exploited. Conventional superscalar processor can not scale well because of the complexity and power consumption of large-issue-width and huge-instruction-window processor. One way to solve this problem is to partition the code into regions, and execute multiple regions speculatively on multiple cores. This method increases both the issue width and the instruction window size dramatically, thus more ILP can be extracted. In such a system, communicating operands between instructions is one of the performance bottlenecks. In addition, 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 [3, 4, 6, 9, 10] 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 com-

pile 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. However, this approach requires ISA support and may not automatically adapt to microarchtectural components such as window size.

Our experimental vehicle is TFlex [7], a composable multicore processor, which implements an EDGE ISA [12]. 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 System Overview

TFlex is a composable lightweight processor in which all microarchitectural structures, including the register file, instruction window, predictors, and L1 caches are distributed across a set of cores [7]. Distributed protocols implement instruction fetch, execute, commit, and misprediction recovery without centralized logic.

TFlex implements an EDGE ISA which supports block-atomic execution. Thus, fetch, completion, and commit protocols operate on blocks rather than individual instructions. The compiler [14] breaks the program into single-entry, predicated blocks of instructions.At runtime, each block is allocated to one core and is fetched into the instruction queue of that core. The union of all blocks running simultaneously on distributed cores

i:	add c, b, a	*i:*	add <j, op1> <k, op2>
j:	sub e, c, d	*j:*	sub <k,op1>
k:	st e, c	*k:*	st

(a) Initial representation (b) Dataflow representation

Figure 1: A baseline code example.

constructs a large contiguous window of instructions. Inter-block communication for long dependences occur through distributed register files using a lightweight communication network [7].

Register and memory communication is used for inter-block communication. Within blocks, instructions run in dataflow order. A point-to-point bypass network performs producer-consumer direct communication using tokens. When an instruction executes, the address of its target is used to directly index the instruction queue.

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 1(a) and 1(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*.

The aforesaid dataflow encoding eliminates the need for an operand broadcast network. When an instruction executes, the address of its target is used to directly index the instruction queue. Because of this direct point-to-point communication, the instruction queue has a simple 128-entry SRAM structure instead of large, power-hungry CAM structures used for instruction queues in superscalar processors. 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 $predicate$.

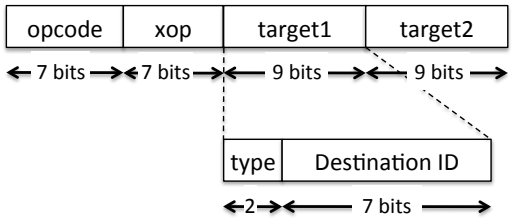

Figure 2: TFlex Instruction Encoding.

Although token based point-to-point communication is very power-efficient for low-fanout instructions but similar to other dataflow machines, it may not be very performance-efficient when running high-fanout instructions since the token needs to travel through the fanout tree to reach all the targets.

3 Hybrid Operand Communication Mechanism

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 three 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.

3.1 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 [11], 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.

3.2 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. 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 [14] 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. 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 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 3 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

i_1:	add c, a, b	i_1:	add <i_2, op1> <i_{1a}, op1>	i_1:	add [SBCID=1, op1]
i_2:	sub e, c, d	I_{1a}:	mov <i_3, op1> <i_4 op1>	i_2:	sub [RBCID=1] <i_5, op1>
i_3:	add f, c, g	i_2:	sub <i_5, op2>	i_3:	add [RBCID=1] <i_5, op1>
i_4:	st d, c	i_3:	add <i_5, op1>	i_4:	st [RBCID=1]
i_5:	st f, e	i_4:	st	i_5:	st
		i_5:	st		

(a) Initial representation (b) Dataflow representation (c) Hybrid dataflow/broadcast representation

Figure 4: A sample code and corresponding code conversions in the modified compiler for the hybrid model.

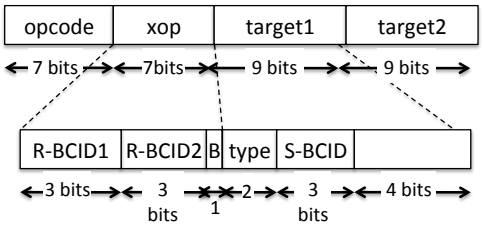

Figure 3: TRIPS Instruction Encoding with Broadcast Support. S-BCID, R-BCID and B represents send BCID, receive BCID and the broadcast enable flag.

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 4 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 4(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 4(c), the compiler 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.

3.3 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 5 illustrates the instruction queue of a single TFlex core when running the broadcast instruction i_1 in the sample code shown in Figure 4(c). When the broadcast instruction executes the broadcast signal, bit B in Figure 3 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 6 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 conitions is true:

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

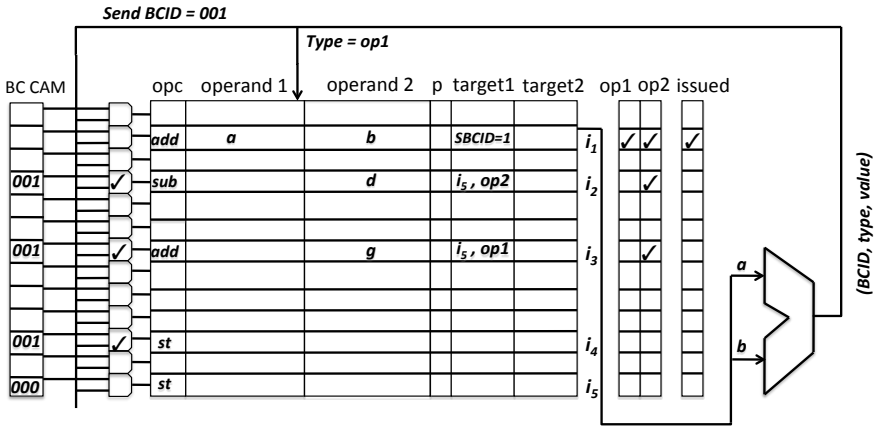

Figure 5: Execution of a broadcast instruction in the IQ.

- 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 5 and 4.

- 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

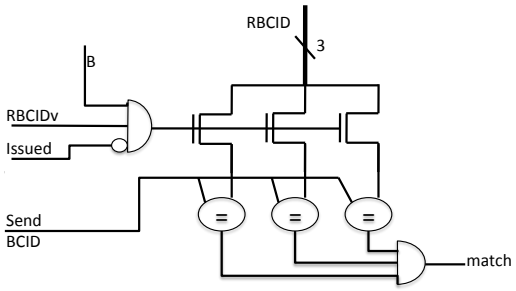

Figure 6: Compare logic of BC CAM entries.

hybrid point-to-point/broadcast communication model [6]. 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 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. These results show

Table 1: Single Core TFlex Microarchitecture Parameters [7]

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 [8] (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

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.

4.1 Methodology

We augment the TFlex simulator [7] 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 1 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 [7]. We augment the baseline TFlex models with the power consumed in the $BCCAM$ entries, modeled using CACTI 4.1 [5], 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 [13]. 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.

4.2 Distribution of Producers and Operands

Figure 7 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 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 8 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 8 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.

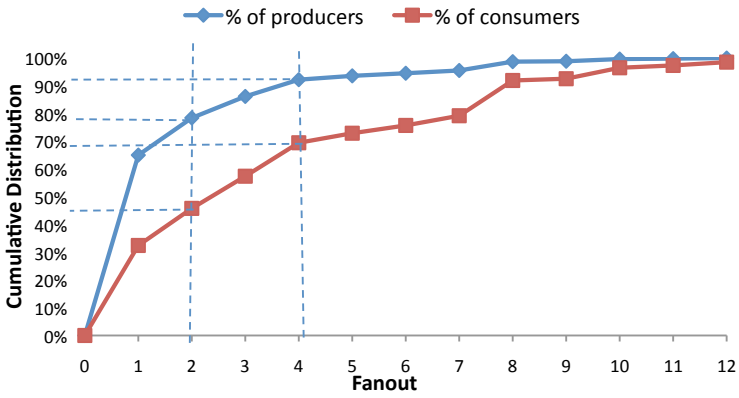

Figure 7: Cumulative distribution of producers and operands.

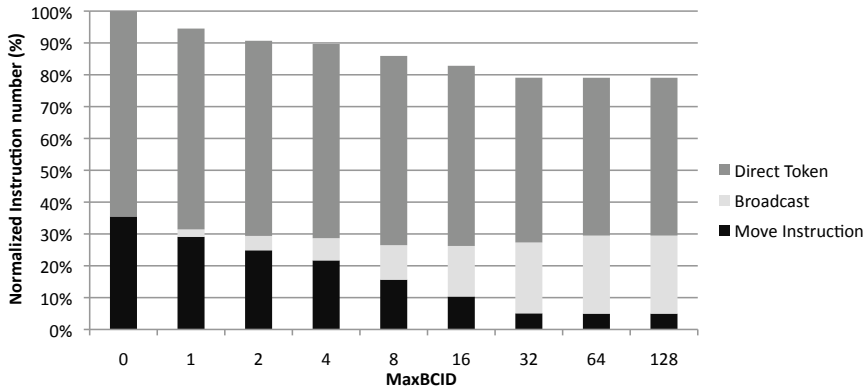

Figure 8: The ratio of broadcast, move and other instructions.

4.3 Energy Tradeoff

Figure 9 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.

Figure 9 also shows the lower bound energy consumption values derived using an analytical model. This analytical model

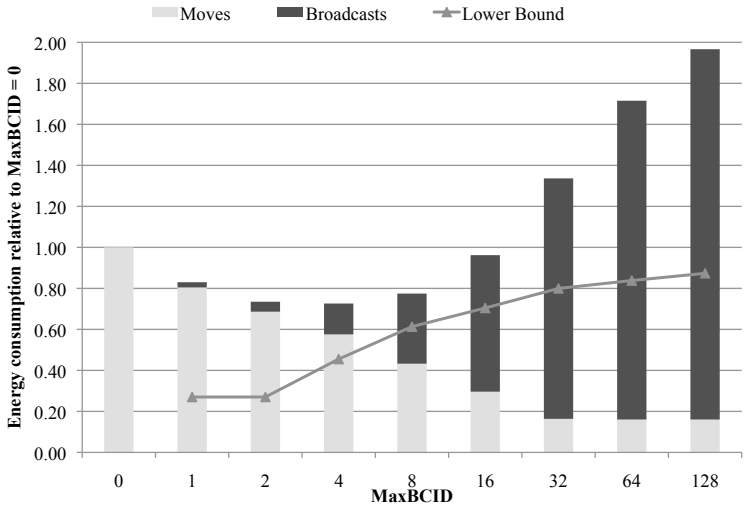

Figure 9: Averaged energy breakdown between move instructions and broadcasts for various MaxBCIDs for SPEC benchmarks.

Max BCID	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 2: Percentage of broadcast producers for real and ideal models.

gives us the best communication mechanism for each producer in an ideal 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 [5] and the power model in the TFlex simulator [7], 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 9, 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 2 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.

4.4 Performance Improvement

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.

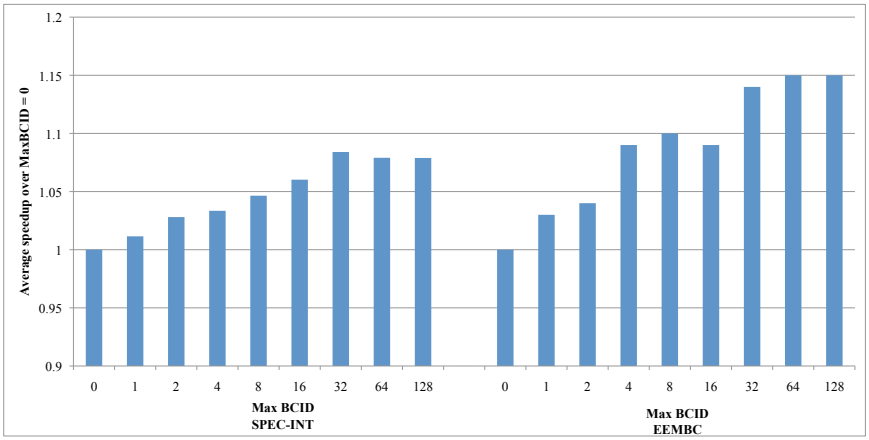

Figure 10: Average speedups achieved using various MaxBCIDs for SPEC and EEMBC benchmarks.

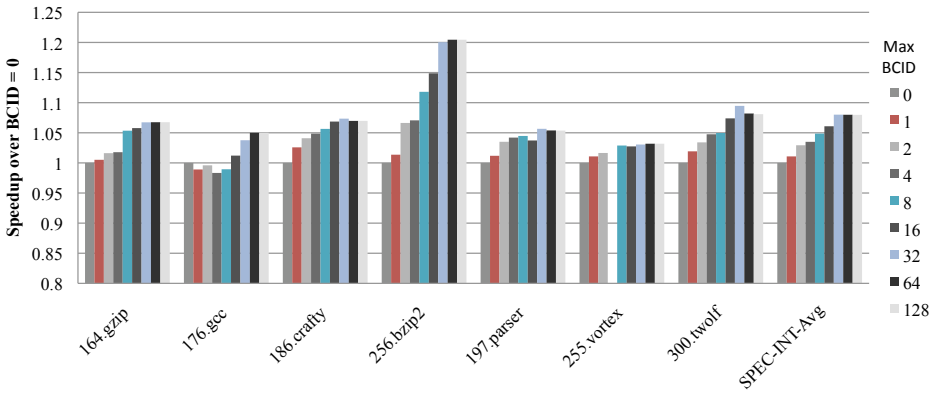

Figure 11: Speedups achieved using various MaxBCIDs for individual SPEC benchmarks.

There is an important tradeoff 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 10 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 8, 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 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 11 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 9 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 28%.

5 Conclusions

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. 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.

References

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.