Appears in the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17)

Exploiting Criticality to Reduce Bottlenecks in Distributed Uniprocessors

Behnam Robatmili[†] *beroy@cs.utexas.edu*
Sibi Govindan[†] *sibi@cs.utexas.edu*
Doug Burger[‡] *dburger@microsoft.com*
Stephen W. Keckler[†§] *skeckler@nvidia.com*

[†]Department of Computer Science, The University of Texas at Austin
[‡]Microsoft Research
[§]Architecture Research Group, NVIDIA

Abstract

Composable multicore systems merge multiple independent cores for running sequential single-threaded workloads. The performance scalability of these systems, however, is limited due to partitioning overheads. This paper addresses two of the key performance scalability limitations of composable multicore systems. We present a critical path analysis revealing that communication needed for cross-core register value delivery and fetch stalls due to misspeculation are the two worst bottlenecks that prevent efficient scaling to a large number of fused cores. To alleviate these bottlenecks, this paper proposes a fully distributed framework to exploit criticality in these architectures at different granularities. A coordinator core exploits different types of block-level communication criticality information to fine-tune critical instructions at decode and register forward pipeline stages of their executing cores. The framework exploits the fetch criticality information at a coarser granularity by reissuing all instructions in the blocks previously fetched into the merged cores. This general framework reduces competing bottlenecks in a synergic manner and achieves scalable performance/power efficiency for sequential programs when running across a large number of cores.

1 Introduction

Due to limitations in clock frequency scaling, chip multiprocessors (CMPs) have become popular to continue scaling performance for explicitly parallel applications. However, delivering performance improvements for single-threaded applications using multicore systems remains a challenging problem. Even for parallel applications, the sequential part of the code can become a serious bottleneck, according to Amdahl's law. Several proposals in recent years have attempted to address these problems. An *asymmetric multicore* (A-CMP) processor has a few high-performance cores for running single-threaded code and several light-weight cores for running parallel code [15]. A-CMPs can work efficiently for some types of workloads but they are not flexible enough to adapt to a wide range of workload characteristics. This lack of flexibility is due to the fixed issue width and execution bandwidth of the large cores allocated to sequential codes.

Another approach to address the need for adapting to different workloads is using composable or dynamic multicores [9, 11]. In these architectures, independent cores can be merged (fused) and share resources to run a single-threaded application. When running a fully parallel application, however, each core can run a single thread. Other configurations are also possible. A recent analytical study [8] shows that compared to other alternatives such as A-CMPs, a performance-scalable composable system can achieve the best power/performance trade-offs by leveraging various degrees of parallelism expected in future workloads. However, the scalability of composable cores is still an open question because they are subject to overheads caused by partitioning the single-threaded code across the distributed substrate. The correct fetch, execute, and commit operations of single-threaded applications must be guaranteed across distributed cores. Register dependences between instructions distributed among the cores must be detected and maintained in a scalable manner.

Prior composable designs have not scaled well as the number of merged cores goes up. For example, TFlex [11] uses an EDGE ISA and takes advantage of compiler-driven block-atomic execution to break the fine-grained register and control dependences and can merge up to 32 dual-issue cores. Most SPEC INT benchmarks reach their maximum performance when running on eight cores and observe a slowdown when running on 16 or more cores. This paper uses critical path analysis [5, 28] to systematically detect and quantify the bottlenecks in such composable, multi-core systems. This technique, which can be used to analyze any architecture, indicates that the most critical bottleneck in the TFlex substrate is the coarse-grained, inter-core register communication, which occurs through shared forwarding units. The second-worse bottleneck is a fetch bottleneck

caused by mispredictions. To alleviate these bottlenecks, this paper evaluates a distributed framework called *Distributed Block Criticality Analyzer* (DBCA) that exploits different types of criticality information collected at block boundaries to implement low-overhead optimizations at fine or coarse execution granularities. This general and flexible framework is implemented in a fully distributed fashion across multiple cores. Such a framework can be used for exploiting different types of criticality to optimize applications dynamically in future distributed systems. Although the proposed framework is general, for the purpose of this study, we focus on the following two types of criticality:

Communication Criticality: DBCA predicts critical communication instructions at block boundaries using a low-overhead criticality predictor located in a coordinator core, which is not necessarily the same as the core executing those instructions. After these critical instructions are predicted, they are selectively optimized according to their criticality types at a pipeline-stage granularity in their executing core using two mechanisms. First, *selective register value bypassing* sends values directly from each output-critical instruction in one executing core to their consumer instructions in other cores, thus bypassing shared register forwarding units. Second, *selective instruction merging* dynamically reduces the length of dependence paths originating from input-critical register values communicated across cores.

Fetch Criticality: To exploit fetch criticality, the coordinator core maintains availability status of previously fetched blocks and reissues those blocks if needed. Consequently, this method saves latency and energy by short-circuiting the fetch and decode operations of all instructions in the reissued block.

The evaluation shows that DBCA reduces the effect of the major bottlenecks simultaneously, thus resulting in a major speedup while reducing power consumption when running across a large number of cores. For instance, using 16 merged cores, DBCA improves performance and energy efficiency (measured in inverse of energy delay squared) of the system by 26% and 68%, respectively for SPEC integer benchmarks.

2 Related Work

CoreFusion [9] is a composable microarchitecture in which groups of two or four dual-issue, out-of-order cores are dynamically fused to form a larger processor. When fused, each core uses its private i-cache and branch predictor to fetch instructions and predict branches. The information about branch prediction decisions must be transferred to a central unit called the fetch management unit to arrange a consistent sequence of executing instructions. Fetched instructions are sent to another centralized unit for register renaming and finally to their executing cores. The use of the physically shared register renaming and fetch units causes bottlenecks and limits the aggregate issue width to eight.

Instead of resolving cross-core data/control dependences dynamically, some approaches take advantage of compiler support to extract instruction dependencies statically. Instruction Level Distributed Processing [12] supports hierarchical register files consisting of many general purpose registers and a few accumulator registers. The hardware steers each compiler-detected strand of instructions to a processing element and its accumulator. The inter-strand dependencies are handled through the general purpose registers. Distributed dataflow-like architectures, including Explicit Dataflow Graph Execution (EDGE) architectures can also support a varying number of dynamic elements assigned to a single thread. TRIPS uses the compiler to form predicated blocks of dataflow instructions and to place each instruction on a 16-ALU grid, where they issue dynamically [23]. TFlex is a second generation EDGE design that supports dynamic core aggregation [11], and is the underlying distributed substrate used in this paper. Multiscalar [26] and Thread-level Speculation [13] rely on discontinuous instruction windows by having the hardware spawn speculative compiler-selected threads on multiple cores.

Fields and Bodik [5] propose a state-of-the-art criticality predictor for superscalar processors. In this predictor, the executing core (processor) detects and sends last-arriving edges of microarchitectural events to the predictor as training data. The predictor uses a relatively high-overhead forward token passing algorithm to detect long-lasting chains of instructions using two parallel training and prediction paths. The criticality predictor presented in this paper, however, is customized for predicting critical register communication instructions between code blocks where each block is running on a separate core and uses a very low-overhead majority vote algorithm [3]. Some methods selectively steer critical instructions differently in the pipeline. Seng et al. [24] propose a microarchitecture that steers the predicted critical instructions to a high-performance/power pipeline while steering the non-critical instructions towards a low-performance/power pipeline. In a distributed processor with light-weight cores, providing multiple pipelines per core could be costly. DBCA uses communication criticality information to steer communication-critical instructions towards different decode or register forward pipeline stages. It also uses fetch criticality to short circuit the fetch and decode of instructions in reissued fetch-critical blocks.

Krishnan and Torrellas [14] propose a hardware-based cross-core register communication in thread-level speculation systems using a synchronizing scoreboard and a shared bus. Restricting register bypassing to immediate successor blocks, DBCA employs a selective critical value bypassing that does not incur any of these overheads. To further reduce

the overhead, it also performs cross-core value bypassing only for the predicted critical register output values.

Several static compiler optimizations such as dynamic move elimination, register integration, and constant folding are inherently limited by the compile scope and ISA restrictions. Runtime optimizations like RENO [21] have proposed dynamic versions of those static optimizations. These mechanisms applied at decode and register renaming can potentially reduce the dependence height of program instruction chains. Applying this mechanism to all fetched instructions can cause a relatively high overhead. DBCA uses a RENO-like decode-time dynamic instruction merging but focuses that mechanism only on critical input instructions.

Trace processors exploit control independence by reusing control-independent traces in the window following misprediction events. The trace generation hardware implements complex algorithms for detecting fine-grain, intra-trace control-independence and coarse-grain, inter-trace global re-convergent points [20, 17]. Taking advantage of the compiler-generated predicated blocks, DBCA employs a block reissue mechanism that does not use these hardware components. Moreover, in this reissue mechanism, each coordinator core only maintains the availability status of its associated blocks, which amortizes the bookkeeping overhead across a large number of instructions. Sankaralingam [22] et al. propose instruction revitalization for TRIPS, in which the compiler adds a setup block to the beginning of each loop to dynamically initiate reissuing of the loop body. The block reissue method used by DBCA leverages the same concept of block revitalization, but it is not limited to loops, is fully dynamic, and requires no statically added setup code.

3 Background

TFlex is a composable lightweight processor that implements an EDGE ISA supporting block-atomic execution [11]. Fetch and commit protocols operate on blocks rather than individual instructions. Within a block, each instruction explicitly encodes its target instructions, and executes when its operands arrive. All branches with intra-block trargets are converted to data dependences using predication. The compiler breaks the program into single-entry, predicated blocks of instructions, similar to hyperblocks [16]. The ISA imposes several restrictions on blocks to simplify the hardware. The maximum size of each block is 128 instructions. Each block can contain up to 32 register reads, 32 register writes, and 32 load or store instructions. The compiler currently achieves about 64 dynamic instructions per block. Figure 1 and Table 1 illustrate the various microarchitectural components of a TFlex core [11]. Each TFlex core has the minimum required resources for running a single block, which include a 128-entry RAM-based instruction queue, a data cache bank, a register file, a branch prediction table, and an instruction (block) cache bank.

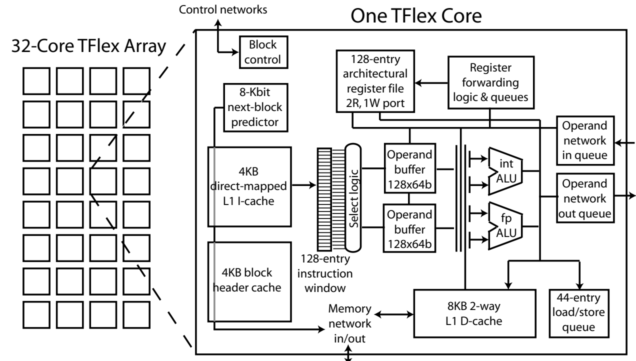


Figure 1. Microarchitectural components of one core of a 32-core TFlex Composable Light-weight Processor.

When *N* cores are merged, they can run *N* blocks simultaneously, of which one block is non-speculative and the rest are speculative. As each block is mapped to the instruction queue of one core, all instructions inside that block execute and communicate within the core [19]. In the merged mode, the register banks, instruction cache banks, and data cache banks of the cores are shared among the cores and are address interleaved. For example, each core contains a data cache and the low-order bits of each memory address determines the core containing the cache bank associated with that memory address [11]. In the merged mode, a register forwarding unit and a load store queue unit on each core are in charge of holding speculative register and memory values produced by the running blocks. Additionally, the register forwarding unit resolves dependences and forwards register values between blocks. Therefore, a register value produced by a block needs to be first sent to its *home* core so that its consuming blocks can be identified and it can get forwarded to the cores running those blocks. Consequently, there is no centralized renaming mechanism for inter-block register communication. Additionally, distributed protocols implement next block prediction, fetch, execute, commit, and misprediction recovery using no centralized logic, which makes this architecture able to scale to 32 cores, in the best case, supporting a single thread. The block-level prediction, fetch, commit, misprediction recovery overheads are amortized across all instructions in each block.

4 Detecting Bottlenecks using Critical Path Analysis

This study addresses the question of where the microarchitectural bottlenecks lie as many cores are merged to-

Table 1. Single Core TFlex Microarchitecture Parameters [11]

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, Call Target Buffer: 16, Branch Target Buffer: 128, Btype: 256.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to 2 INT and 1 FP).
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 [10] (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.

gether to accelerate a single thread. Critical path analysis [5, 28] is an effective way to study the interactions among microarchitectural events and to identify the bottlenecks in a processor system. The simulation-based critical path analysis used in [5] generates and processes the program dependence graph. This dependence graph is a directed acyclic graph, where nodes represent the various microarchitectural events and edges represent data or microarchitectural dependences among these events. Microarchitectural dependences are dictated by the characteristics of the microarchitectural components of the target processor such as branch predictor, fetch, issue, commit and memory units. In the critical path analysis, events (nodes) are tagged by their corresponding resource(s). Summing the delays associated with all nodes with a same tag, this analysis can estimate the critical path contribution of the component associated with that tag. The system simulator [11] outputs a trace of the various microarchitectural events that occur during the execution of a program. Each event in this trace includes all of the data needed to be added to the dependence graph including the cycle of the event occurrence and the block associated with the event. The critical path tool then constructs the program dependence graph using the trace and computes the program critical path according to the algorithm in [18]. The tool reports the critical path breakdown which includes the contribution of each microarchitectural component to the critical path. Although the tool reports several critical path components, most of the critical cycles are spent in one of the following components:

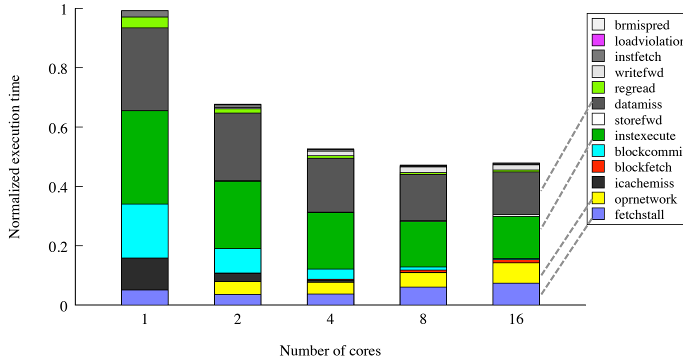
(1) **branch misprediction**: branch misprediction overhead. (2) **load violation**: load dependence misspeculation overhead. (3) **data misses**: the time spent on data misses. (4) **instruction execution**: the execution time spent in ALUs. (5) **network**: the time spent on operand communication across the network. (6) **fetch stalls**: the time waiting for a fetch-critical block to be fetched. (7) **block commit**: the time spent when a block has finished executing all its instructions and waits for the previous blocks to commit before it can commit. Other components reported by this tool include *instruction fetch*, *write forward* and *store forward*. *Write forward* and *store forward* are the times spent in register files or load/store queues when a register or memory value is forwarded between speculative blocks, respectively.

Figure 2 reports the critical path breakdown for SPEC-INT and SPEC-FP benchmarks. Different stacked bars represent different system configurations. Each system configuration is associated with a fixed number of merged cores. Each segment of the critical path is normalized against the corresponding segment length in the 1-core configuration. Therefore, the total height of the stack for each configuration represents the average execution time in that configuration normalized against the execution time in the 1-core configuration. The dotted lines in the figure highlight the major components in the critical path. For INT benchmarks, when running on only one core, the dominant bottlenecks in the system are *execution bandwidth*, *block commit*, and *data misses*. As more cores are merged, execution bandwidth, block commit bandwidth, and data cache capacity/bandwidth are increased. Consequently, *execution, block commit* and *data misses* become less critical. This trend continues for the configurations with core counts smaller than eight. When using eight or more cores, *data misses, instruction execution, on-chip inter-core network* and *fetch stalls* are the major contributors of the critical path. Among these factors, the contributions of the *on-chip network* and *fetch stalls* (the lowest two segments in each configuration shown) increase as more cores are merged. The following two components can be considered as the system's main bottlenecks:

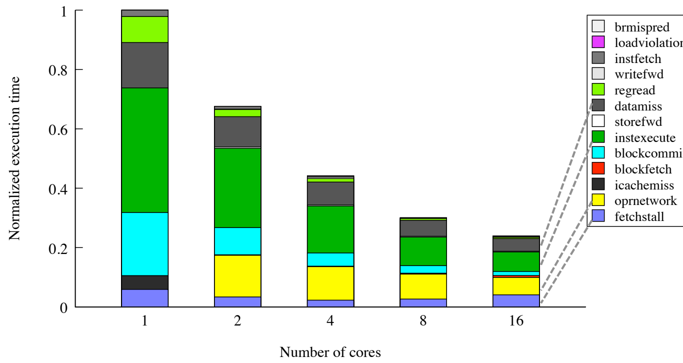
On-chip Network: The on-chip network is used for communication between instructions running on different merged cores. Our results show that most of the network traffic is caused by register communication between distributed instructions. A producer instruction sends a new value of an architectural register to the home core of that register. Resolving dependences, the register forwarding unit of that core then forwards the value to its consumer core(s). When the core count increases, the network distance increases and so does the average inter-block communication delay.

Fetch: When a large number of cores are merged, the system constructs a large window of speculative instructions. Consequently fetch stalls caused by misspeculation flushes are likely to end up on the critical path and become a performance bottleneck. This phenomenon explains the increase in fetch stall segments of the critical path when merging more cores.

If these bottlenecks were eliminated many of the critical cycles would shift to execution. FP benchmarks scale better than INT benchmarks and on chip network is not as critical

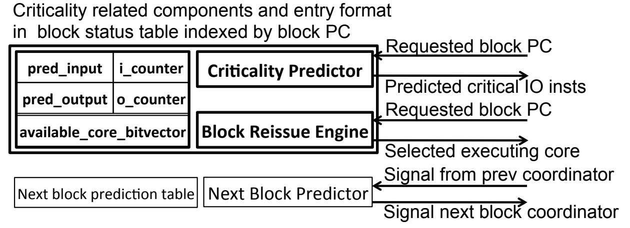


(a) INT

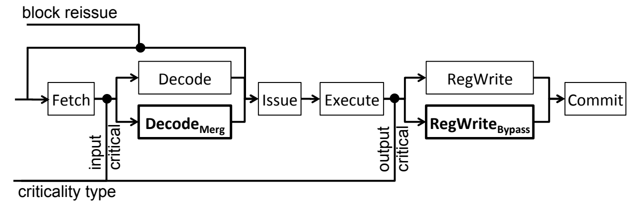


(b) FP

Figure 2. Critical path breakdown of *SPEC* benchmarks for different microarchitectural components.



(a) Tables and components added for coordinating.



(b) Augmented instruction pipeline.

Figure 3. Components used in the distributed block criticality analyzer to reduce bottlenecks.

for them but fetch stalls are still on their critical path.

5 Distributed Block Criticality Analyzer

This section explains a distributed block criticality analyzer (DBCA) that exploits criticality information to optimize critical instructions or code blocks based on their criticality characteristics. In this paper, DBCA is restricted to cross-core communication and fetch criticality. However, it can be extended to include other types of criticality such as memory and execution criticality. Figure 3 highlights the components added to each TFlex core by this analyzer. To minimize the communication overhead, DBCA piggybacks on the next block prediction distributed protocol. Each block is assigned a fixed core as its coordinator core, which is selected based on a few low-order bits of the PC of the first instruction in the block (block PC). The coordinator core contains next block prediction tables for all of the blocks assigned to it. When a new block is requested, its coordinator core is signaled by the coordinator of the previous block to allocate an idle core (a core not executing any block) to execute the new block and predict the next block and then signal the coordinator core of the predicted block.

DBCA extends this protocol by augmenting the coordinator core with a table called *the block status information* table shown in Figure 3(a). This table contains different criticality information of blocks assigned to this coordinator core and is maintained by two hardware components located on the coordinator core. A communication criticality predictor predicts both the communication-critical instructions and their *criticality type* and a block reissue component maintains the information required for reissuing the non-running instances of fetch-critical blocks. When a block is allocated, the corresponding coordinator core accesses these hardware components, extracts and sends the criticality information of that block to the selected executing core. The executing core uses that information to optimize the pipeline of critical instructions according to their criticality types. Communication critical instructions are treated specially in a fine-grained manner in the pipeline according to their predicted criticality type. Output-critical instructions go through a value bypassing stage which sends the produced critical register values directly from their producer cores to their consumer cores, thus bypassing the shared register forwarding units. Input-critical instructions go through a decode time dynamic instruction merging stage that reduces the height of their dependent instruction chains. For reissued fetch-critical blocks, all instructions skip their fetch and decode stages in a coarse-grained manner.

Note that in this distributed framework, coordinator and executing cores do not have to be physically separated and a core executing a block can simultaneously act as the coordi-

Table 2. An example of mapping 4 loop iterations each with 2 blocks A and B, across 8 cores (C_1 to C_7).

Fetch order Block$_{iteration}$	A_1	B_1	A_2	B_2	A_3	B_3	A_4	B_4
Coordinator cores	C_0	C_1	C_0	C_1	C_0	C_1	C_0	C_1
Executing cores	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7

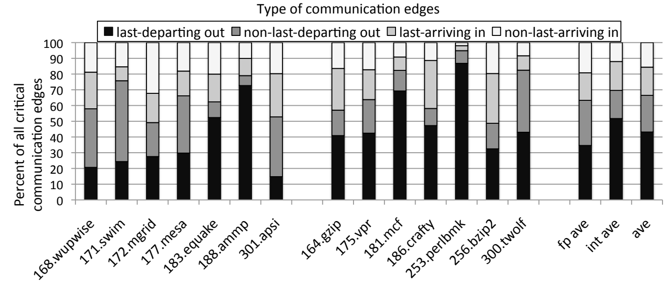


Figure 4. Critical-communication edges breakdown for SPEC benchmarks.

nator core of other blocks. Table 2 shows coordination and execution orders in a system running 4 iterations of a loop across 8 cores. Each iteration has 2 blocks A and B (block PCs) assigned to coordinator cores C_0 and C_1, respectively. If all cores are idle at first, the coordinator cores select idle cores in a round-robin fashion for running the iterations.

5.1 Communication Criticality Predictor

Predicting critical communication instructions of each block can be done using a state-of-the-art criticality predictor [5] explained in Section 2. Although proven effective [5], using such a criticality predictor in a distributed multicore system can cause high hardware complexity, communication, and storage overheads. In this subsection, we describe a low-overhead communication criticality predictor used by DBCA. For each block, *block inputs* refer to the register operands used by instructions in that block, but produced by other blocks. *Block outputs*, on the other hand, refer to the register operands produced by the instructions in that block but used by other blocks in the window. As long as all inputs of a block have not arrived from previous blocks, some instructions in that block remain uncompleted. Finally, the block cannot commit until all its outputs are sent to other blocks. Therefore, late communication edges (last-departing register outputs produced by a block before the block commits or the last-arriving register inputs received by a block) are likely to be on the critical path. To verify this, we use the critical path analysis discussed in Section 4 to find the breakdown of the critical communication edges. In this breakdown, the critical communication edges are divided into four categories: last-departing outputs, non-last-departing outputs, last-arriving inputs and non-last-arriving inputs. Figure 4 presents this breakdown for the SPEC2K benchmarks running across 16 merged cores (a few benchmarks are missing due to critical path tool complications). Note that the critical output and input edges are the edges on the critical path from a producing core to the corresponding forwarding unit on the home core of the corresponding register and from the forwarding unit to a consuming core, respectively. For INT benchmarks, 70% of all register critical inputs and outputs are late communication edges (the sum of the first and third segments from below in each bar). For FP benchmarks, late communication edges may be less critical because only 52% of critical register inputs and output are late.

Given the high criticality of the late communication edges, to reduce overheads, the predictor used by DBCA predicts late communication edges instead of critical communication edges. We explain the algorithm for predicting the last-arriving register inputs of each block; predicting last-departing outputs is similar. The coordinator core stores late input predictions for its assigned blocks ($pred_input$ in Figure 3(a)). For critical register inputs, the actual predicted value is the register number associated with the last-arriving register input of the block. When a block is allocated and mapped to a core, its coordinating core predicts the last-arriving register input of that block and sends it to the core executing that block. When the block ends execution, its executing core appends the last-arrived input observed during execution to a *dealloc* message and sends it to the coordinator core along with other data needed for deallocation. The coordinator core updates the prediction entry of that block using Boyer and Moore's majority vote algorithm [3]. Each entry in the table includes the predicted last-arriving input for a block and a majority vote counter ($i_counter$ in Figure 3(a)). When the predictor updates the entry, if the new last-arriving input is the same as the predicted one, the majority counter is incremented. Otherwise, the counter is decremented but the predicted input does not change. If the counter reaches zero, the predicted input will be updated by the current last-arriving input. To reduce the effect of the stale data in the prediction table, the predictor uses an epoch-based algorithm. This algorithm uses two prediction entries per block, each with an input number and a majority counter. During each fixed epoch, the algorithm uses one of the two entries for training and the other entry that was trained in the previous epoch, for predicting. When an epoch ends, the two entries are switched. Our accuracy evaluation of this predictor shows that when running the SPEC benchmarks across 16 cores, late register inputs and outputs of blocks can be predicted correctly 80% of the time.

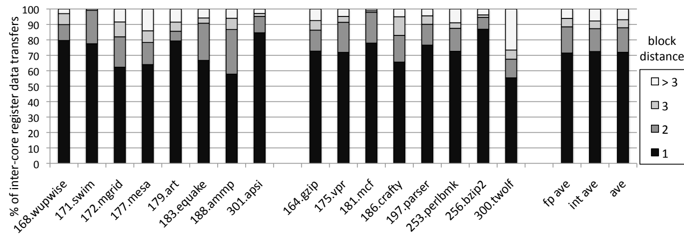


Figure 5. Block distances between register data producer and consumer blocks running on 16 merged cores.

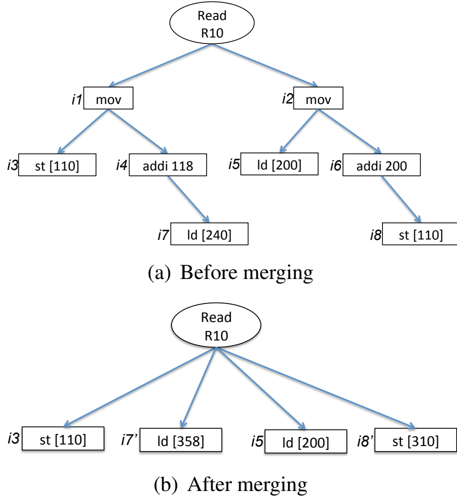
5.2 Selective Register Value Bypassing

Before we discuss the register bypassing used by executing cores for critical instructions, we briefly discuss the the original register forwarding mechanism used by TFlex. The original mechanism uses distributed forwarding units in participating cores to resolve inter-block instruction register dependences. As each speculative block is allocated, it allocates an entry for each of its output registers in the forwarding unit of the *home core* of that register. Also, it sends its register read requests to corresponding home cores of its input registers. When a block produces a register output, the core running the block sends the output value to the home core of the corresponding register. In the home core, the register forwarding unit accesses the forwarding entry allocated to that register to look up the destination cores and instructions before sending the value. In this mechanism, the inter-block register dependences are resolved at the home core. This forwarding mechanism is *indirect* because it does not provide direct communication between source and destination cores. The communication delay associated with this forwarding increases as the network grows larger. An alternative communication model uses direct communication between different blocks. In this model, register dependences are resolved in the producer cores and each producer instruction in one core sends it output directly to the consumer instruction(s) in other cores. Such a mechanism requires that each core maintain a synchronizing scoreboard table [14] that tracks the status of all shared registers. To keep the tables updated and coherent, cores need to be connected though a shared broadcast bus, which can reduce scalability as high numbers of cores are used.

As shown in Figure 3(b), DBCA uses a low-overhead, direct communication mechanism called *value bypassing* for critical output instructions. However, other instructions use the original indirect forwarding mechanism. We restrict register value forwarding to only immediate successive speculative blocks. Figure 5 shows block forwarding distances between producer and consumer blocks for the SPEC benchmarks running on 16 cores. The block forwarding distance in this figure is the number of speculative blocks between a producer block and its consumer blocks. On average, 74% of the value forwarding happens between two subsequent speculative blocks. Considering the aforementioned simplifications, the bypass mechanism for critical registers no longer needs to track the status of all registers using synchronizing scoreboards. When the last-departing register of a block is predicted, the core executing the block sets a flag if the subsequent block reads that register. When the last-departing value is produced, the producer core sends the value directly to the core executing the next speculative block if the flag is set. The destination core forwards the value to its instructions waiting for the that register value. The value also needs to be sent to the home core of that register so that non-critical consumer cores can also receive it through the original forwarding mechanism.

5.3 Selective Instruction Merging

As shown in Figure 3(b), during decode, the executing cores use an instruction merging mechanism. This mechanism works like RENO [21] in general but is only applied to the instructions consuming a predicted critical input. This mechanism tracks instructions consuming the predicted last-arriving register and removes the *move* and *add-immediate* instructions that generate the address for a load or store instruction. The mechanism merges the immediate values into the offset operand of the destination load or store instruction and tags the load or store instruction as a new immediate consumer of the critical register input. The tracking process ends after detecting any non-copy or a non-add-immediate instructions in the dependence chain of the critical register. During execution, when the value of the predicted critical register is received, it will be directly sent (broadcast) to all its consumer instructions detected during decode. Figure 6(a) shows a part of the instruction dependence graph in a block. For simplicity, the figure only shows the dependences related to address calculation for memory instructions and data-related dependences are not shown. Register *R10* is used by two stores, which are *i3* and *i8*, and two load instructions, which are *i5* and *i7*, as their base address. The base address is incremented by two immediate values using add-immediate instructions before it reaches *i7* and *i8*. There are two copy (or move) instructions, which are *i1* and *i2*, starting the base address distribution network. Figure 6(b) shows the instruction dependence graph of the same code after the instruction merging mechanism is applied at decode. The copy and add-immediate instructions are eliminated and their immediate values are added to the offsets of their consuming memory instructions to form new instructions *i7'* and *i8'*. Finally, all of the memory instructions are tagged as consumers of the the critical input register *R10*. During execution, once the value of *R10* arrives, the value is sent (broadcast) as the base address to all of the



(a) Before merging

(b) After merging

Figure 6. An instruction dependence graph before and after instruction merging.

tagged memory instructions which are *i3, i5, i7'* and *i8'*.

5.4 Block Reissue

The block reissue component in the framework tracks and reissues instances of blocks previously-executed in the distributed instruction window. Different from trace processors [20, 17], this *block reissue* mechanism does not rely on complex hardware for tracking and combining the blocks in flight and finding global re-convergent points. Instead, it relies on the compiler to detect re-convergent points and create large predicated blocks by combining basic blocks. Normally all instructions in each block remain in the instruction window of the executing core until the block commits or is flushed. Distributed instruction queues on the participating cores can be used as intermediate instruction storage. Shortcutting critical fetch and decode operations as shown in Figure 3(b), the mechanism achieves energy savings. For example, when the first iteration of the loop in Table 2 commits, A_1 and B_1 available instances of blocks A and B in the instruction queue can be immediately reissued to run iteration 5 on cores C_0 and C_1.

To support block reissue, a coordinator core stores a bit vector ($available_cores_bitvector$ in Figure 3(a)) for each block assigned to it, which represents the idle cores in which a non-running copy of the block is available. When a block is allocated/committed, its coordinator core resets/sets the bit corresponding to the executing core of that block. When a block is requested, its coordinator core searches the corresponding bit vector to find a core with a non-running instance of the block, reissues that block, and resets the corresponding bit in its bit vector. If the block is not available in any of the idle cores, the executing core selects an idle core to fetch the block from the i-cache and execute it. In this case, the selected core deallocates its previously-executed block and informs the coordinator core of that block to update the bit vector of the deallocated block. To increase the hit rate of the block reissue mechanism, we can use more than one instruction queue in each core. For instance, when using two instruction queues per core, each core can store up to two decoded blocks. At a given time, however, each core can only execute one of its two stored blocks and the other instruction queue is used as an instruction storage.

6 Results

This section evaluates individual and aggregated mechanisms used by the distributed block criticality analyzer. We modified a cycle accurate TFlex simulator [11] to add the support for DBCA. We accurately model the distributed protocols and all the components used by DBCA (see Figure 3). Table 1 shows the microarchitectural parameters for each TFlex core used in our experiments. Each TFlex core is a dual-issue, out-of-order core with a 128-entry instruction window for executing one block. At the same time, it acts as the coordinator core for a set of blocks. We also augment the original TFlex power models to accurately model the energy consumed/saved by the components in DBCA. The original TFlex power models are developed similar to the Wattch power models [4], but are constructed using validated TRIPS power models [6]. Due to space limitations, we refer the reader to the original TFlex paper [11] for a discussion of the power modeling methodology. We use the CACTI [27] tool to augment the original power model with accurate models of the hardware tables and components highlighted in Figure 3. We use eight integer and eight floating point the SPEC2K [1] benchmarks using the reference (large) dataset simulated with single SimPoints, version 3.2 [25]. Except for scalability charts, we report 16-core configuration results because the maximum improvements are observed in that configuration. A complete cross-platform comparison is beyond the scope of this paper, but is available in [7].

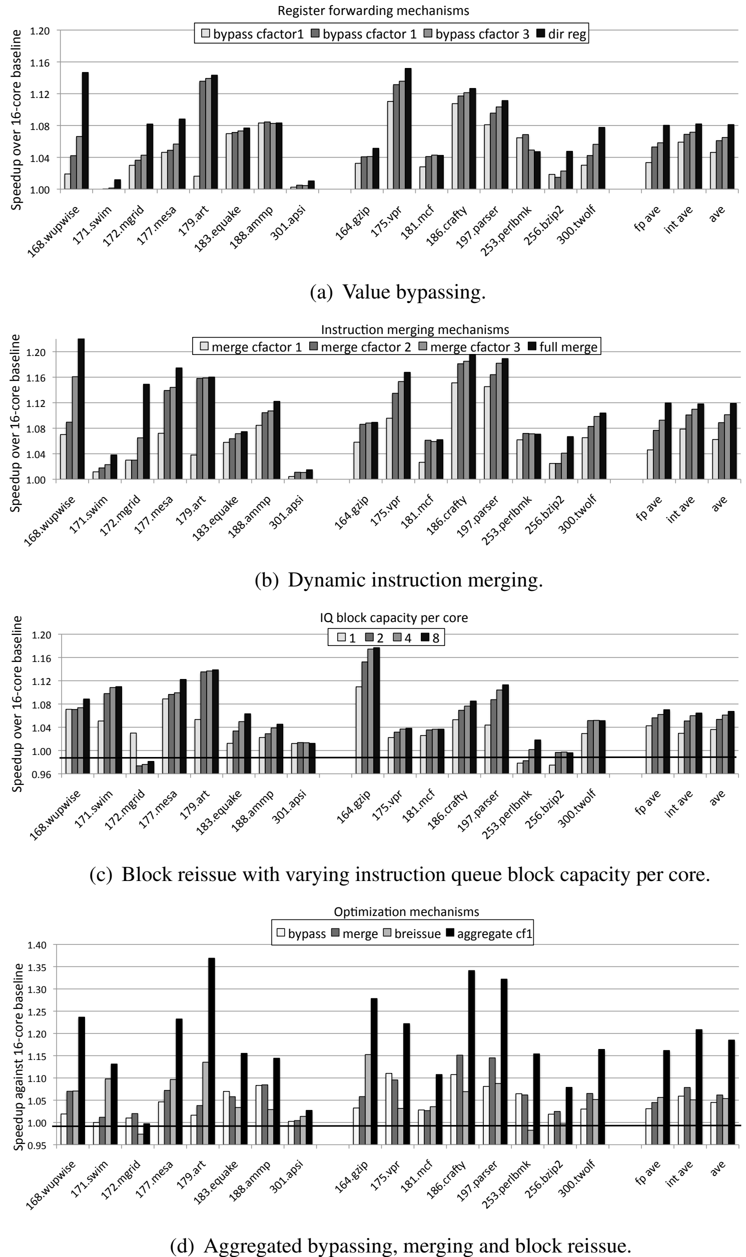
6.1 Individual Components

Figure 7(a) shows the speedups achieved using selective register value bypassing for the SPEC benchmarks when using 16 merged cores. To control the number of forwarded critical values per block, we use a parameter called *criticality factor*. This factor determines the number of forwarded last-departing register values per block. For instance, using a criticality factor of three means that every block bypasses its three last-departing register outputs to the core running the subsequent speculative block. In this graph, *bypass cfactor 1* to *3* represent the selective bypass runs

with criticality factors of 1 to 3. *Dir reg* represents a high-overhead register forwarding mechanism in which all producers forward their values directly to their consumers in all consecutive blocks. *Dir reg* can be used as an upper limit for measuring value bypassing speedups. For the SPEC INT benchmarks, the maximum speedup over the original bypassing model on average is 8%, which is achieved using dir reg. Only bypassing one last-departing register output of each block will achieve about 6% speedup on average. This speedup increases to more than 7% as we raise the criticality bypass factor to 3. For FP benchmarks, the maximum speedup is about 8%, while using criticality factor of 1 results in 3% speedup. Raising the criticality factor to 3 increases the speedup to 6%. As shown in Figure 4, the last-departing register values are not as critical for FP benchmarks as they are for INT benchmarks.

Figure 7(b) illustrates the speedups achieved using selective critical instruction merging for the SPEC benchmarks when merging 16 cores. The criticality factor in this figure determines the number of last-arriving registers per block, used by the merging algorithm during the decode stage. *Merge cfactor 1* to *3* represent the selective instruction merging mechanism with criticality factors of 1 to 3. To provide an upper limit for measuring speedups, *full merge* represents a hypothetical, aggressive instruction merging mechanism in which all copy and add-immediate instructions following each input register are merged into their destinations. For the SPEC INT benchmarks, the maximum speedup is more than 12%, which is achieved using full merge. Applying instruction merging only to the most critical register input of each block (merge cf 1) achieves more than a 7% speedup over the baseline which does not use merging. The speedup increases to 11% as we change the criticality factor from 1 to 3. In contrast, for FP benchmarks, the speedup changes from 4% to 8% when varying the criticality factor from 1 to 3. This difference between the INT and FP benchmarks again indicates the fact that last-arriving inputs are less critical for FP benchmarks as shown in Figure 4.

For the block reissue experiments, we implement an LRU block replacement policy and a first-match search on the available_cores_bitvector bit vector of each allocated block. More complex search and replacement policies can improve reissue the hit rate of fetch-critical blocks. Figure 7(c) reports the 16-core execution times normalized against a 16-core baseline which does not use block reissue. For some benchmarks, we observe a slowdown when applying block reissue. For these benchmarks, block reissue has a negative effect on the accuracy of the data dependence predictor. Ignoring those benchmarks, average speedups across INT and FP benchmarks for different block storage size per core are very similar. When storing one block in the instruction queue of each core, the average reissue hit



(a) Value bypassing.

(b) Dynamic instruction merging.

(c) Block reissue with varying instruction queue block capacity per core.

(d) Aggregated bypassing, merging and block reissue.

Figure 7. 16-core speedups achieved using individual and by aggregated components.

rate is about 50%, which translates to decreasing the energy consumed by fetch and decode by half. The issue hit rate increases to 60%, 67% and 75% when using instruction queue block capacity of 2, 4 and 8, respectively. The reissued blocks come from different sources on the INT and FP benchmarks. Table 3 includes the breakdown of reissued blocks for the runs with instruction queue block capacity of two. For INT benchmarks, the majority of the reissued blocks are reissued after block misprediction events. For FP

Table 3. Percentage breakdown of reissued blocks with IQ block capacity of two.

Benchmarks	After branch mispredictions	After load violations	Single block loops	others
INT	54.4%	20.0%	1.9%	23.7%
FP	26.0%	9.8%	11.9%	52.3%

benchmarks, large loops and other repetitive code patterns comprises the majority of reissued blocks.

6.2 Aggregated Distributed Block Criticality Analyzer

Each of the three mechanisms evaluated so far is used to shorten a segment of the critical path reported in Figure 2(a) and when applied alone, produces a mild performance improvement for INT benchmarks. This could be due to competing bottlenecks in the system. For instance, reissuing a fetch-critical block helps performance as long as that block has not become communication critical. Also, running selective bypassing and merging on a communication-critical block will improve performance as long as the block has not become fetch-critical. Figure 2(a) shows that the major bottlenecks such as inter-core communication and fetch bottleneck grow as the number of merged cores goes up. Consequently, when only the effect of one of them is reduced, the other bottleneck is more exposed on the critical path and prevents high speedups. Figure 7(d) shows the 16-core speedups for each optimization mechanism when applied alone and when these mechanisms are aggregated. For selective bypassing and merging reported in this graph, the criticality factor is set to 1. For block reissue, each core can hold two blocks in its instruction queue, one of which is executing at a time. For INT benchmarks, each individual mechanism achieves an speedup of about 5% to 7% while when aggregated, these mechanisms achieve an average speedup of 22% over the baseline that does not use DBCA. By using criticality factor of two and three, which is not shown in this graph, the average speedup increases to 24% and 26%, respectively.

This significant speedup is caused by simultaneous reduction of the major system bottlenecks. Figure 8 shows the breakdown of the critical path for INT benchmarks before and after using DBCA. This figure only includes configurations with more than four merged cores and uses a 1-core configuration as its baseline. The stacked bars labeled *base* and *aggreg_cf1* show the criticality breakdown before and after applying DBCA, respectively. In this figure, the optimizations employed by DBCA use criticality factor of one. The communication and fetch stall segments of the critical path (the two lowest segments of each stack) are significantly reduced after applying DBCA. This reduction can explain the large speedup achieved by aggregating mechanisms. The new critical path comprises mostly instruction execution and data cache misses. This criticality pattern is nearly ideal for a system with distributed partitions of a large instruction window. According to this graph, most useful cycles of a program execution are now spent on instruction execution and data misses. A more detailed evaluation of the critical path indicates the execution is now limited only by program data dependencies, including intra-block predicates and next block prediction accuracy. Therefore, even further performance can be squeezed by extending DBCA for execution criticality and branches, which is ongoing research.

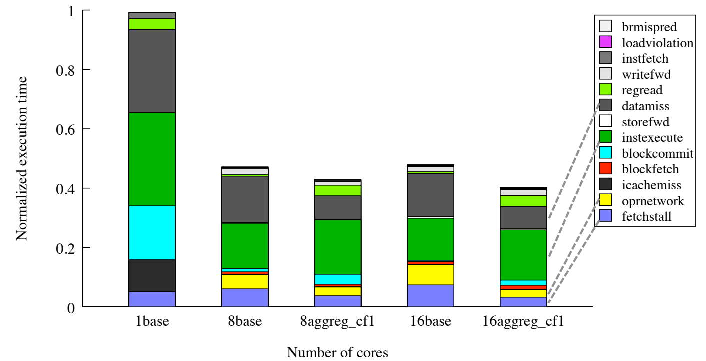


Figure 8. Critical path breakdown of INT benchmarks for different microarchitectural components for the baseline (base) and after using the distributed block criticality analyzer (aggreg_cf1).

6.3 Performance/Energy Scalability

Figure 9 illustrates the average system performance when varying the number of merged cores and optimization mechanisms. These numbers are speedups over single dual-issue cores for the SPEC INT and FP benchmarks. The different charts in the figure are as follows. **baseline:** This original TFlex system does not use DBCA and its components. **bypass:** In this mode, selective critical bypassing is enabled with criticality factor of one. **bypass_merge:** This mode uses selective bypassing and merging with criticality factor of one. **aggregate cf 1:** In this mode, selective bypassing and merging are enabled with criticality factor of one and block reissue is enabled with two blocks in each core's instruction queue. **aggregate cf 3:** The same as *aggregate cf 1* except that it uses a criticality factor of 3.

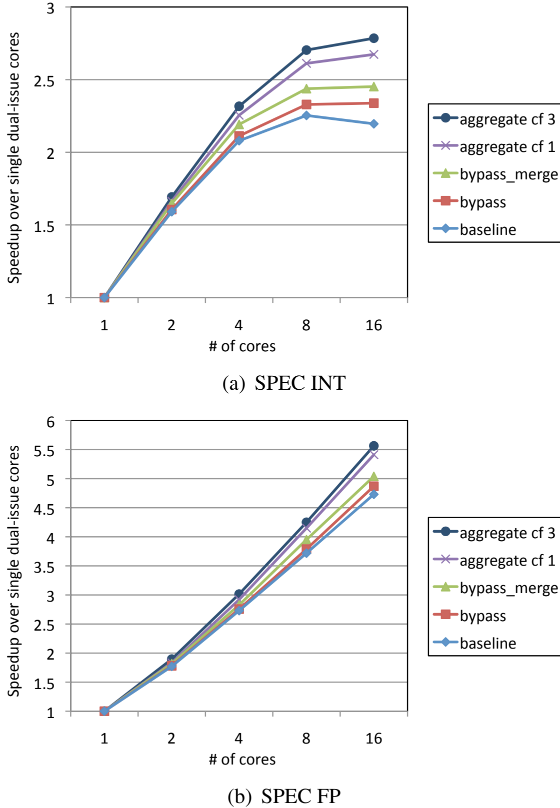
As expected, the 2-core and 4-core speedups are relatively small given the short inter-communication distances and the number of blocks predicted. However when applying the proposed mechanisms using the criticality factor of one, the 8-core average speedup improves from 2.25 to 2.7

Table 4. 16-core percentage energy costs of different optimization mechanisms.

	Register bypass	Dynamic merging	Block reissue lookup and replacement	Block reissue saved fetch & decode	Total saved energy
INT	0.6%	1.6%	0.7%	-9.3%	-6.4%
FP	0.3%	1.0%	0.4%	-6.8%	-5.1%

for INT benchmarks. The average speedup for the 16-core runs improves from 2.2 to 2.8. In addition to improving overall system performance, DBCA also improves the scalability of the system. For instance, for INT benchmarks, the 8-core configuration runs outperform the 16-core configuration runs when none of the optimizations is active. Once all optimizations are active, the 16-core runs outperforms the 8-core runs. FP benchmarks illustrate better overall scalability. Most FP benchmarks are compute and data intensive and will take advantage of duplicated processing elements and data cache banks as more cores are merged.

We also study the energy overhead and efficiency of DBCA. The energy results assume process technology of 65nm, supply voltage of 1.0 V, and a core frequency of 1 GHz. Table 4 shows the energy consumption or savings of each of the proposed mechanism normalized against total energy dissipated in the processor. For INT benchmarks, the three mechanisms raise the processor's consumed energy by 2.9%. Block reissue, however, reduces total energy by 9.3% by shortcutting fetch and decode operations, resulting in total energy savings of 6.4%. For FP benchmarks, this number is 5.1%. The 16-core configuration witnesses highest performance/energy improvement measured in the inverse of energy-delay^2 of about 68%. For FP benchmarks, the 16-core configuration also shows the highest performance/energy improvement of about 36%.



(a) SPEC INT

(b) SPEC FP

Figure 9. Average speedup over single core for the SPEC benchmarks with varying numbers of merged cores and optimizations.

7 Conclusions

This paper addresses several key performance scalability limitations of distributed, composable multicore systems. The critical-path based analysis shows that the communication needed for register dependence resolution among distributed instructions and supporting a consistent fetch stream are the two major bottlenecks that occur when merging high numbers of cores. To alleviate these bottlenecks, this study proposes a flexible framework for exploiting different types of instruction criticality in a distributed dynamic multicore system. This framework called distributed block criticality analyzer or DBCA, augments each core with several very low-complexity distributed components and implements a distributed protocol to optimize different types of critical instructions at different levels of pipeline across cores. Critical communication instructions are predicted in this framework and optimized at a pipeline-stage granularity. Critical output instructions use a fast register forward stage for communicating their results to remote cores while critical input instructions use a high performance decode stage that reduces their dependence height. Finally, at the block level, DBCA implements a mechanism that reissues blocks of instructions while they are still in the instruction window.

This framework can be implemented on other distributed systems and can exploit other criticality types. Moreover, coarse-grained grouping of related instructions at compile-time similar to the one used in this study can simplify the implementation and reduce its overheads. Our results show DBCA can significantly improve power/energy scalability of the system. For example, DBCA achieves performance and energy efficiency improvements of 26% and 68%, respectively for 16-core sequential runs compared to an unoptimized 16-core system. Also, the optimized system using DBCA in most configurations shows performance scalability close to that expected by Pollack's rule (e.g. 2.7x across 8 cores) [2]. According to this rule, uniprocessor performance increases proportional to square root of increase in complexity or area [2].

Acknowledgements

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

References

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

[2] S. Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference*, pages 746–749, June 2007.

[3] R. S. Boyer and J. S. Moore. MJRTY - a fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe, of Automated Reasoning Series*, pages 529–543, 1977.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Computer Architecture News*, 28(2):83–94, 2000.

[5] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture*, pages 74–85, July 2001.

[6] M. S. S. Govindan, S. W. Keckler, and D. Burger. End-to-end validation of architectural power models. In *International Symposium on Low Power Electronics and Design*, pages 383–388, San Francisco, September 2009.

[7] M. S. S. Govindan, B. Robatmili, H. Esmaeilzadeh, B. Maher, D. Li, A. Smith, S. W. Keckler, and D. Burger. Scaling power and performance via processor composability. Technical report, 2010. UT Austin, Department of Computer Sciences TR-10-14.

[8] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. In *IEEE Computer*, volume 41, pages 33–38, July 2008.

[9] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, June 2007.

[10] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

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

[12] H.-S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *International Symposium on Computer Architecture*, pages 71–81, May 2002.

[13] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, September 1999.

[14] V. Krishnan and J. Torrellas. The need for fast communication in hardware-based speculative chip multiprocessors. *International Journal of Parallel Programing*, 29(1):3–33, 2001.

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *International Symposium on Microarchitecture*, pages 81–93, December 2003.

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

[17] S. W. Melvin, M. C. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *Workshop on Microprogramming and Microarchitecture*, pages 60–63, November 1988.

[18] R. Nagarajan, X. Chen, R. McDonald, D. Burger, and S. Keckler. Critical path analysis of the TRIPS architecture. In *International Symposium on Performance Analysis of Systems and Software*, pages 37–47, March 2006.

[19] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *International Symposium on Microarchitecture (MICRO)*, pages 23–34, November 2008.

[20] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *International Symposium on Microarchitecture*, pages 138–148, December 1997.

[21] E. Rotenberg and J. Smith. RENO: a rename-based instruction optimizer. In *International Symposium on Computer Architecture*, pages 98–109, June 2005.

[22] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger. Universal mechanisms for data-parallel architectures. In *International Symposium on Microarchitecture*, pages 303–314, December 2003.

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

[24] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *International Symposium on Microarchitecture*, pages 114–123, December 2001.

[25] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, June 2001.

[26] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 521–532, June 1995.

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

[28] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *International Symposium on High Performance Computer Architecture*, pages 181–195, January 2001.