Composable Lightweight Processors

Changkyu Kim† [*] Simha Sethumadhavan M.S. Govindan Nitya Ranganathan
Divya Gulati Doug Burger Stephen W. Keckler

† Microprocessor Technology Labs
Intel Corporation
changkyu.kim@intel.com

Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu

Abstract

Modern chip multiprocessors (CMPs) are designed to exploit both instruction-level parallelism (ILP) within processors and thread-level parallelism (TLP) within and across processors. However, the number of processors and the granularity of each processor are fixed at design time. This paper evaluates a flexible architectural approach, called Composable Lightweight Processors (or CLPs), that allows simple, low-power cores to be aggregated together dynamically, forming larger, more powerful single-threaded processors without changing the application binary. We evaluate one such design with 32 cores called TFlex, which can be configured as 32 dual-issue processors, or as a single 64-wide issue processor, or as any point in between. Use of an Explicit Data Graph Execution (EDGE) ISA enables the system to be fully composable, with no monolithic structures spanning the cores. Simulation results show that CLPs achieve an average performance boost of 42%, an average area-efficiency of 3.4x, and an average power-efficiency of 2x over a fixed architecture on a spectrum of single-threaded applications. Results also show that CLPs outperform a spectrum of fixed CMP architectures on a set of multitasking workloads.

1 Introduction

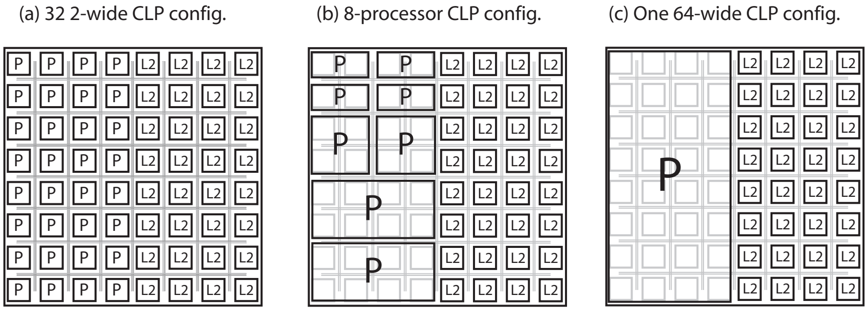
Due to limitations on clock frequency scaling, most future computer system performance gains will come from power-efficient exploitation of concurrency. Consequently, the computer industry has migrated toward chip multiprocessors (CMPs), in which the capability of the cores depends on the target market. Some CMPs use a greater number of narrow-issue, in-order cores (Niagara [15]), while others use a smaller number of out-of-order superscalar cores with SMT support (IBM Power5 [29]). One disadvantage of this approach for non-server domains is that enough software threads must be found to utilize all of the processors. An additional disadvantage of this conventional CMP approach is its relative inflexibility. In a conventional design, the granularity (i.e., issue width) and number of processors on each chip are fixed at design time, based on the designers' best analyses about the desired workload mix and operating points. Any such fixed design point will result in suboptimal operation as the number and type of available threads change over time.

In this paper, we evaluate an alternative CMP design called Composable Lightweight Processors (or CLPs) to eliminate the problem of fixed-granularity processors. A CLP consists of multiple simple, narrow-issue processor cores that can be aggregated dynamically to form more powerful single-threaded processors. Thus, the number and size of the processors can be adjusted on the fly to provide the target that best suits the software needs at any given time. The same software thread can run transparently—without modifications to the binary—on one core, two cores, up to as many as 32 cores in the design that we simulate. Low-level run-time software can decide how to best balance thread throughput (TLP), single-thread performance (ILP), and energy efficiency. Run-time software may also grow or shrink processors to match the available ILP in a thread to improve performance and power efficiency.

Figure 1 shows a high-level floorplan with three possible configurations of a CLP. The small squares on the left of each floorplan represent a single processing core while the squares on the right half show a banked L2 cache. If a large number of threads are available, the system could run 32 threads, one on each core (Figure 1a). If high single-thread performance is required and the thread has sufficient ILP, the CLP could be configured to use an optimal number of cores that maximizes performance (up to 32, as shown in Figure 1c). To optimize for energy efficiency, for example in a data center or in battery-operated mode, the system could configure the CLP to run each thread at its best energy-efficient point. Figure 1b shows an energy-optimized CLP configuration running eight threads across a range of processor granularities.

A fully composable processor shares no structures physically among the multiple processors. Instead, a CLP relies on distributed microarchitectural protocols to provide the

[*] Work done while a Ph.D. student at the University of Texas at Austin


(a) 32 2-wide CLP config.

(b) 8-processor CLP config.

(c) One 64-wide CLP config.

Figure 1: Three dynamically assigned CLP configurations.

necessary fetch, execution, memory access/disambiguation, and commit capabilities. Full composability is difficult in conventional ISAs, since the atomic units are individual instructions, which require that control decisions be made too frequently to coordinate across a distributed processor. Explicit data graph execution (EDGE) architectures, conversely, reduce the frequency of control decisions by employing block-based program execution and explicit intra-block dataflow semantics, and have been shown to map well to distributed microarchitectures [6]. The particular CLP design that we evaluate, called TFlex, achieves the composable capability by mapping large, structured instruction blocks across participating cores differently depending on the number of cores that are running a single thread.

This paper compares various configurations of the TFlex CLP to a fixed-granularity TRIPS processor, which uses the same ISA. The TFlex CLP microarchitecture allows the dynamic aggregation of any number of cores–up to 32 for each individual thread–to find the best configuration under different operating targets: performance, area efficiency, or energy efficiency. The performance, area, and power models are derived from and validated using the TRIPS hardware. On a set of 26 benchmarks, including both high- and low-ILP codes, results show that the best configurations range from one to 32 dual-issue cores depending on operating targets and applications. The TFlex design achieves a 1.4x performance improvement, 3.4x performance/area improvement, and 2.0x performance^2/Watt improvement over the TRIPS processor. The TFlex CLP also shows improved parallel flexibility with the throughput of multiprogrammed workloads improving by up to 47% over the best fixed-CMP organization.

2 Related Approaches

The goal of effectively adapting parallel resources to varying number of threads is a long-standing one. The recent Core Fusion [12] work is most similar to the CLP approach. Like CLPs, Core Fusion allows multiple dynamically allocated processors to share a single contiguous instruction window. The advantage of Core Fusion is that it exploits conventional RISC or CISC ISAs. Because of this advantage, some structures (e.g. register renaming) must be physically shared, limiting its scalability to 8-wide issue. The TFlex CLP described in this paper shares no resources physically, so it can scale up to 64-wide issue, but relies on a non-standard EDGE ISA to achieve full composability.

WaveScalar [32], a tagged-token dynamic dataflow architecture, provides the same capability of running single-threaded applications on multiple processing elements (PEs) or up to as many threads as the number of PEs, but requires different distributed protocols and structures since WaveScalar supports no explicit speculation.

Most other prior work uses either *partitioning* or *composition* to provide adaptive parallel granularity [25]. Of the prior work that employs composition–synthesizing a large logical processor from smaller processing elements–most uses independent sequencers with a non-contiguous instruction window. An early example is Multiscalar [31]. Later Thread-Level Speculation designs employed processors in a CMP [10, 16]. CLPs differ from such architectures in that they employ a single logical point of control, i.e. a contiguous instruction window, across the multiple processing elements, which simplifies dependence tracking.

Instead of using a compositional approach to balance ILP and TLP dynamically, Simultaneous Multithreading–in which multiple threads share a single large, out-of-order core–instead uses a partitioning approach [33]. The OS achieves low-overhead, adaptive granularity by adjusting the number of threads that are mapped to one processor. The disadvantage is that the range of granularity is limited, since processors have limited issue width and threads sharing the same core may cause significant interference, or the aggressive processors may be overkill when only one limited-concurrency thread is available.

Conjoined-core chip multiprocessing falls in between composition and partitioning, with some structures shared and others distributed [18]. Other approaches have provides statically exposed architectures that can be partitioned. Compiling to distributed targets, as in RAW [34], an important and early tiled architecture, allows statically varied degrees of composition. Decoupled Software pipelining (DSWP) [21] uses the compiler to extract TLP from single-threaded application loops. The Voltron compiler [35] can futher extract parallelism from single-threaded applications by exploiting both VLIW-style ILP and fine-grain TLP. In statically exposed architectures, however, the application must be recompiled for each new configuration.

Other work has focused on adapting individual structures to balance performance and energy, rather than varying the number of processors participating in running a thread [2]. This approach has been extended to dynamically resizing the caches [1], issue window [8], load/store queue and register file [22], and issue width [3]. Finally, single-ISA heterogeneous CMPs include a discrete number of fixed processors of different granularities that cannot be composed, but which present a range of granularity options to the scheduler [17]. This approach increases design complexity and limits the number of granularity options, but avoids adaptation and composability overheads.

3 Instruction-Set Support for Composability

Creating larger logical microarchitectural structures from smaller ones is the principal challenge for the design of a composable architecture. Composing some structures, such as register files and level-one data caches, is straightforward as these structures in each core can be treated as address-interleaved banks of a larger aggregate structure. Changing the mapping to conform to a change in the number of composed processors merely requires adjusting the interleaving factor or function.

However, banking or distributing other structures required by a conventional instruction set architecture is less straightforward. For example, operand bypass (even when distributed) typically requires some form of broadcast, as tracking the ALUs in which producers and consumers are executing is difficult. Similarly, instruction fetch and commit require a single point of synchronization to preserve sequential execution semantics, including features such as a centralized register rename table and load/store queues. While some of these challenges can be solved by brute force, supporting composition of a large number of processing elements can benefit from instruction set support.

The TFlex architecture avoids the challenges of distributing individual instructions by using an Explicit Data Graph Execution (EDGE) ISA designed specifically for a distributed microarchitecture [6]. Although EDGE ISAs were not originally designed for composability, their two major distinguishing features align them well with the requirements of CLPs.

First, EDGE ISAs encode programs as a sequence of blocks which have atomic execution semantics. This block-atomic model allows the control protocols for instruction fetch, completion, and commit to operate on large blocks—128 instructions in the TRIPS ISA—instead of individual instructions. This amortization reduces the frequency of control decisions, reduces the overheads of bookkeeping structures and makes the core-to-core latencies tractable [25]. For example, since only one branch prediction is made per block, a prediction in TRIPS is needed at most once every eight cycles. This slack makes it possible for one core to make a prediction and then send a message to another core to make a subsequent prediction without significantly reducing performance.

Second, each EDGE instruction in a block explicitly encodes which dependent instructions should receive its result. In a distributed architecture, this encoding eliminates the need for an operand broadcast bus, since a point-to-point network can interpret the identifiers as coordinates of instruction placement. In a composable architecture, differently sized "logical" processors merely interpret the target instructions' coordinates differently for each composed processor size.

4 Microarchitectural Support for Composability

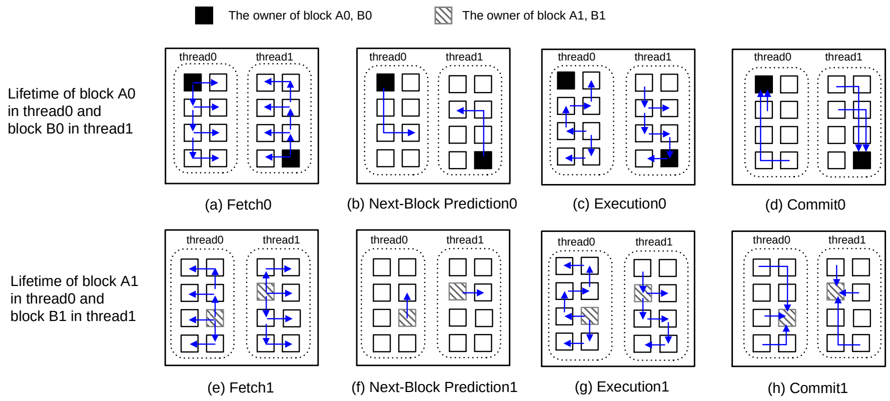
A composable processor must allow its microarchitectural structures to linearly increase (or decrease) in capacity as participating cores are added (or removed). For example, doubling the number of cores should double the number of useful load/store queue entries, the usable state in the branch predictors, and the cache capacities. The CLP microarchitecture partitions structures by address whenever possible, and avoids physically centralized microarchitectural structures completely.

This complete partitioning addresses some of the limitations of the original TRIPS microarchitecture. Specifically, the next-block predictor state and the number of data cache banks were limited by the centralization of the predictor and the load-store queue, respectively. Full composability necessitates distributing those structures as well, which provides higher overall performance than the TRIPS microarchitecture irrespective of the composable capabilities. However, those performance gains are a side benefit to the significantly increased flexibility that composition provides.

The TFlex microarchitecture uses three distinct hash functions for interleaving across three classes of structures.

Block starting address: The next-block predictor resources (e.g., BTBs and local history tables) and the block tag structures are partitioned based on the starting virtual address of a particular block, which corresponds to the program counter in a conventional architecture. Predicting control flow and fetching instructions in TFlex occurs at the granularity of a block, rather than individual instructions.

Instruction ID within a block: A block contains up to 128 instructions, which are numbered in order. Instructions are interleaved across the partitioned instruction windows and instruction caches based on the instruction ID, theoret-


The owner of block A0, B0
The owner of block A1, B1
Lifetime of block A0 in thread0 and block B0 in thread1
Lifetime of block A1 in thread0 and block B1 in thread1
thread0
thread1

(a) Fetch0 (b) Next-Block Prediction0 (c) Execution0 (d) Commit0

(e) Fetch1 (f) Next-Block Prediction1 (g) Execution1 (h) Commit1

Figure 2: TFlex Execution Stages: Execution of two successive blocks (A0, A1) and (B0,B1) from two different threads executing simultaneously on a 16-core TFlex CLP with each thread running on 8 cores.

ically permitting up to 128 cores each holding one instruction from each block.

Data address: The load-store queue (LSQ) and data caches are partitioned by data address from load/store instructions, and registers are interleaved based on the low-order bits of the register number.

In addition, register names are interleaved across the register files. However, because a single core must have 128 registers to support single-block execution, register file capacity goes unused when multiple cores are aggregated. Because interleaving is controlled by bit-level hash functions, the number of cores that can be aggregated to form a logical processor must be a power of two.

4.1 Overview of TFlex Operation

While a portion of each in-flight instruction block is assigned to each participating core, a block is assigned a single *owner core*, based on a hash of the block address, which is responsible for initiating fetching of the block and predicting the next block. Once the next-block address is predicted, the owner core sends that address to the core that owns the next predicted block. Each owner core is also responsible for launching pipeline flushes of misspeculations caused by its block, for detecting that its block is complete, and for then committing it.

Figure 2 provides an overview of TFlex execution for the lifetime of one block. It shows two threads running on eight cores each. In the block fetch stage, the block owner accesses the I-cache tag for the current block and broadcasts the fetch command to the I-cache banks in the participating cores (Figure 2a). In parallel, the owner core predicts the next block address and sends a control message to the next block owner to initiate fetch of the subsequent block (Figure 2b). Up to eight blocks may be in flight for eight participating cores. Upon receiving a fetch command from a block owner, each core fetches its portion of the block from its local I-cache, and dispatches fetched instructions into the issue window. Instructions are executed in dataflow order when they are ready (Figure 2c). When a block completes, the owner detects completion, and when it is notified that it holds the oldest block, it launches the block commit protocol, shown in Figure 2d. Figures 2e-h show the same four stages of execution for the next block controlled by a different owner; fetch, execution, and commit of the blocks are pipelined and overlapped. Finally, the diagrams show that two distinct programs can be run on non-overlapping subsets of the cores.

4.2 Composable Instruction Fetch

In the TFlex microarchitecture, instructions are distributed across the private I-caches of all participating cores, but fetches are initiated by each of the instruction block's owner core. The owner core manages the tags on a per-block basis, and functions as a central registry for all operations of the blocks it owns. The cache tags for a block are held exclusively in the block owner core. In a 32-core configuration, each core caches four instructions from a 128-instruction block.

With this fetch model, the fetch bandwidth and the I-cache capacity of the participating cores scale linearly as more cores are used. While this partitioned fetch model amplifies fetch bandwidth, conventional microarchitectures cannot use it because current designs require a centralized analysis point of the fetch stream (for register renaming and inum assignment) to preserve correct sequential semantics. This constraint poses the largest challenge to composable processors built with conventional ISAs. In contrast, EDGE ISAs overcome these deficiencies by explicitly and statically encoding the dependence order of the instructions within a block; the dynamic total order among all executing instructions is obtained by concatenating the block order and the statically encoded order within a block. Given large block sizes, distributed fetching is feasible because instruction dependence relations are known *a priori*.

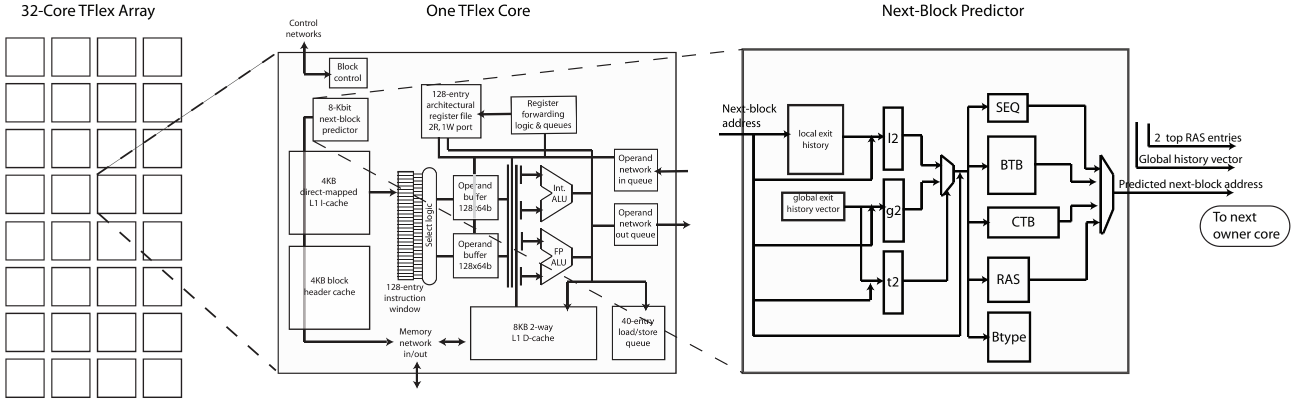


Figure 3: Microarchitectural components of single TFlex core and internal organization of the next-block predictor

4.3 Composable Control-flow Prediction

Control-flow predictors are some of the most challenging structures to partition for composability, since the predictor state has traditionally been physically centralized to facilitate few cycles between successive predictions. The TFlex composable predictor treats the distributed predictors in each composed core as a single logical predictor, exploiting the block-atomic nature of the TRIPS ISA to make this distributed approach tenable. Similar to the TRIPS prototype microarchitecture, the TFlex control flow predictor issues one next-block prediction for each 128-instruction hyperblock—a predicated single entry, multiple exit region—instead of one prediction per basic block.

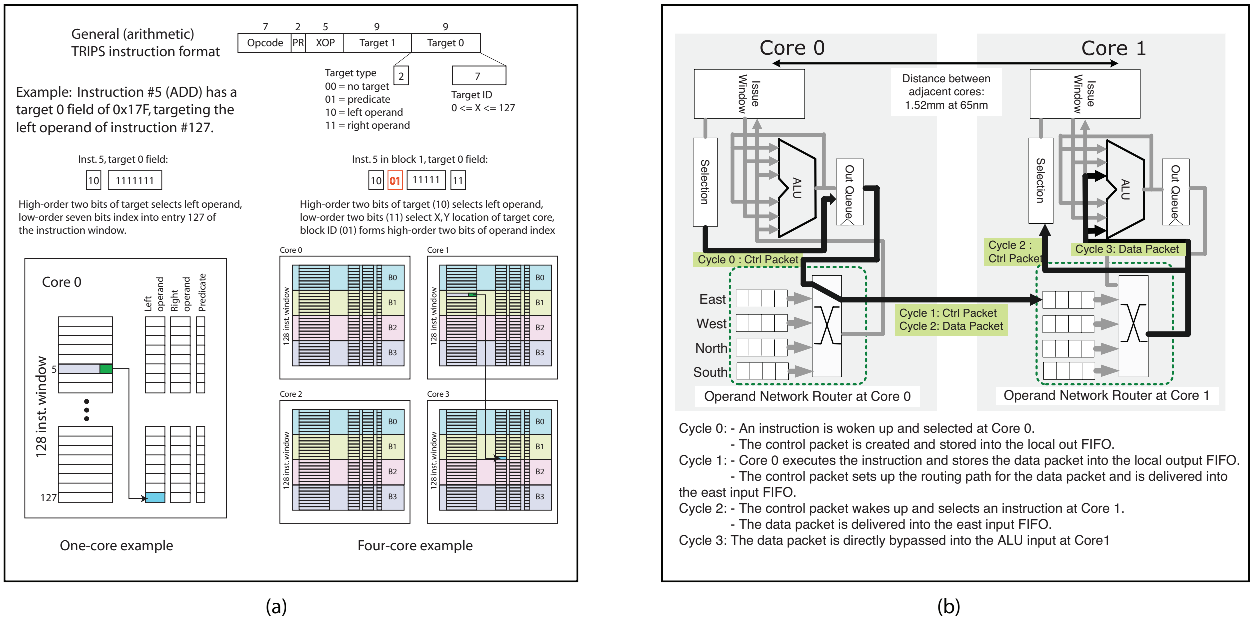
Figure 3 illustrates the distributed next block predictor consisting of eight main structures. Each core has a fully functional block predictor and the predictors are identical across all the cores. The block predictor can be divided into two main components: the exit predictor, predicting which branch will be taken out of the current block; and the target predictor, which predicts the target address of the next block. Each branch in a block contains three *exit* bits in its instruction, which are used to form histories instead of the traditional taken/not taken bits.

The exit predictor is an Alpha 21264-like tournament-style hybrid predictor [13] and is composed of traditional two-level local, global, and choice predictors that use local and global block exit histories. The local as well as global histories are updated speculatively after a prediction and repaired from backup history buffers on a misprediction. The target address is predicted by first predicting the type of the exit branch—a call, a return, a next sequential block, or a regular branch—using the Btype predictor. The Btype predictor result selects one of those four possible next-block targets, which are provided by a next-block adder (SEQ), a Branch Target Buffer (BTB) to predict branch targets, a Call Target Buffer (CTB) to predict call targets, and a Return Address Stack (RAS) to predict return targets.

Each of the major predictor structures is affected by the composed nature of the microarchitecture. The local histories are trivially composable since, for a fixed composition, the same block address will always map to the same core. Local predictions do not lose any information even though they are physically distributed. Similarly, the Btype, BTB and CTB tables hold only the target addresses of the blocks owned by that core. With this organization, the capacity of the predictor increases as more cores are added, assuming that block addresses tend to be distributed among cores equally.

The global predictor is more complex because of the distributed history. When a predicted next-block address is sent from the previous blocks' owner to the owning core of the predicted block, the global exit history is forwarded along with the prediction. This forwarding enables each prediction to be made with the current global history, without additional latency beyond the already incurred point-to-point latency to transmit the predicted next-block address from core to core.

The component that is most difficult to distribute, the Return Address Stack (RAS), must be maintained as a single logical stack across all the cores since it represents the program call-stack. The TFlex microarchitecture uses the composed entries to permit a deeper RAS. Instead of using address interleaving, it sequentially partitions the RAS across all the cores (e.g. a 32-entry stack for 2 cores would have entries 0 to 15 in core 0 and 16 to 31 in core 1). The stacks from the participating cores form a logically centralized but physically distributed global stack. If the exit branch type is predicted as a call, the corresponding return address is pushed on to the RAS by sending a message to the core holding the current RAS top. If the branch is predicted as a return then the address on the stack is popped off by sending a pop-request to the core holding the RAS top. Recovery upon a misprediction is the responsibility of the mispredicting owner, which rolls back the mis-speculated state and sends the updated histories and RAS pointers to the next owner core, as well as the corrected top-of-stack RAS information to the core that will hold the new RAS top.



Example: Instruction #5 (ADD) has a target 0 field of 0x17F, targeting the left operand of instruction #127.

High-order two bits of target selects left operand, low-order seven bits index into entry 127 of the instruction window.

High-order two bits of target (10) selects left operand, low-order two bits (11) select X, Y location of target core, block ID (01) forms high-order two bits of operand index

Cycle 0: - An instruction is woken up and selected at Core 0.
- The control packet is created and stored into the local out FIFO.
Cycle 1: - Core 0 executes the instruction and stores the data packet into the local output FIFO.
- The control packet sets up the routing path for the data packet and is delivered into the east input FIFO.
Cycle 2: - The control packet wakes up and selects an instruction at Core 1.
- The data packet is delivered into the east input FIFO.
Cycle 3: The data packet is directly bypassed into the ALU input at Core1

Figure 4: (a) Block mapping for one-core and four-core processors, (b) Inter-core operand communication

4.4 Composable Instruction Execution

Each instruction in an EDGE block is statically assigned an identifier between 0 and 127 by the TRIPS compiler. A block's instructions are interleaved across the cores of a composable processor using a mapping function. Changing the mapping function when cores are added (or removed), achieves composability of instruction execution. Figure 4a shows the mechanism that the TFlex design uses to support dynamic issue across a variable number of composed cores. Each instruction in an EDGE ISA block contains at least one nine-bit *target* field, which specifies the location of the dependent instruction that will consume the produced operand. Two of the nine bits specify which operand of the destination instruction is targeted (left, right, or predicate), and the other seven bits specify which of the 128 instructions is targeted. Figure 4a shows how the target bits are interpreted in single-core mode, with instruction five targeting the left operand of instruction 127. In single-core mode, all seven target identifier bits are used to index into a single 128-instruction buffer.

Figure 4a also shows how the microarchitecture interprets the target bits when running in a four-core configuration. The four cores can hold a total of four instruction blocks, but each block is striped across the four participating cores. Each core holds 32 instructions from each of the four in-flight blocks. In this configuration, the microarchitecture uses the two low-order bits from the target to determine which core is holding the target instruction, and the remaining five bits to select one of the 32 instructions on that core. The explicit dataflow target semantics of EDGE ISAs make this operation simple compared to what would be required in a RISC or CISC ISA where the dependences are unknown until an instruction is fetched and analyzed.

The latency incurred in routing dependent operands from core to core influences performance greatly. The TFlex cores are connected by a two-dimensional mesh network; bypassing an operand between adjacent cores incurs only a single-cycle bubble as a control message is sent one cycle in advance of the data message to wake up the target instruction. Figure 4b shows the datapath from the output of an ALU in one core to the input of an ALU in an adjacent core and illustrates cycle-by-cycle activities when the execution result at core 0 is bypassed into core 1. Area estimates for 65nm indicate a core-center to core-center distance of 1.5mm, corresponding to an optimally repeated wire delay of 170ps. With a fast router that matches the wire delay, the total path delay would be less than 350ps, enabling a one-cycle inter-core hop latency to be supported at over 2.5GHz.

4.5 Composable Memory System

As with clustered microarchitectures [20, 24], L1 data caches in a composable processor can be address partitioned and distributed into each core. When running in one-core mode, each thread can access only its own bank. When multiple cores are composed, the L1 cache becomes a cache-line interleaved aggregate of all the participating L1 caches. With each additional core, each running thread obtains more L1 D-cache capacity and an additional memory port. The cache bank accessed by a memory instruction is determined by XORing the high and low portions of the virtual address modulo the number of participating cores. All addresses within a cache line will always map to the same bank in a given configuration. When a core computes the effective address of a load, the address and the target(s) of the load are routed to the appropriate cache bank, the lookup is performed, and the result is directly forwarded to the core

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).
Data Supply	Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 44-entry LSQ bank; 4MB decoupled S-NUCA L2 cache [14] (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.
Simulation	Execution-driven simulator validated to be within 7% of real system measurement
Hand-optimized Benchmarks	3 kernels (conv, ct, genalg), 7 EEMBC benchmarks (a2time, autocore, basefp, bezier, dither, rspeed, tblook), 2 Versabench (802.11b, 8b10b) [23]
Compiled Benchmarks	14 SPEC CPU benchmarks currently supported (8 Integer, 6 FP), simulated with single simpoints of 100 million cycles [28].

Table 1: Single Core TFlex Microarchitecture Parameters

containing the target instruction of the load.

One of the microarchitectural challenges to support a composable memory system is to handle memory disambiguation efficiently on a distributed substrate with a variable number of cores. The TFlex microarchitecture relies on load-store queues (LSQs) to disambiguate memory accesses dynamically. As more cores are aggregated to construct a larger window, more entries in the LSQ are required to track all in-flight memory instructions. Partitioning LSQ banks by address and interleaving them with the same hashing function as the data caches is a natural way to build a large, distributed LSQ. However, unless each LSQ bank is maximally sized for the worst case (the instruction window size), a system must handle the situation when a particular LSQ bank is full, and cannot slot an incoming memory request (called “LSQ overflow”). TFlex uses a low-overhead mechanism proposed by Sethumadhavan et al. [27], specifically the NACK mechanism, to handle overflows efficiently.

4.6 Composable Instruction Commit

Cross-core commit overheads in TFlex are low, because the TRIPS ISA requires that instructions within a block are committed en masse. The commit process consumes four phases. First, the block owner detects that a block is complete when participating cores inform the owner core that the block has emitted all of its outputs–stores, register writes, and one branch. Second, when the block becomes the oldest one, the block owner sends out a *commit* command on the control network to the participating cores. Third, all distributed cores write their outputs to architectural state, and when finished, respond with *commit acknowledgment* signals. Finally, the owner core broadcasts a resource deallocation signal, indicating that the youngest block owner can initiate its own fetch and overwrite the committed block with a new block.

4.7 Coherence Management

For a level-two (L2) cache organization, we consider a 4MB shared design (shown in Figure 1) that contains 32 cache banks connected by a switched mesh network. To maintain coherence among private L1 caches, the shared L2 cache uses a standard on-chip directory-based cache coherence protocol with L1 sharing vectors stored in the L2 tag arrays. The composition of the cores on the chip does not affect how the coherence protocol is implemented. A sharing status vector in the L2 tag keeps track of L1 coherence by treating each L1 cache as an independent coherence unit. When a composition changes–adding cores to some composed processors and removing them from others–the L1 caches need not be flushed; the new interleaved mapping for the composed L1 D-cache banks will result in misses, at which point the underlying coherence mechanism forwards the request to the lines stored in the old L1 cache banks, invalidating them or forwarding a modified line.

5 Experimental Methodology

To explore the benefits of flexibility and composability, we compare the composable TFlex architecture to the fixed TRIPS architecture on single-threaded applications using metrics of performance, area, and power. We then examine the overheads associated with distributed protocols for fetch and commit. Finally, we employ a set of multiprogrammed workloads to measure the benefits of composability on throughput. These analyses use a validated cycle-accurate simulator with a set of both compiler-generated and hand-optimized benchmarks shown in Table 1. For the TFlex configurations, the programs are scheduled assuming a 32-core composable processor. Our experience shows that performing instruction scheduling for a larger number of cores and running it on fewer cores results in little performance degradation.

Baseline: We chose the TRIPS processor as the baseline to compare against TFlex for three reasons. First, because TRIPS and TFlex share the same ISA and software infrastructure, their microarchitectures are comparable without needing to compensate for ISA and system level artifacts. Second, TRIPS is a natural baseline because, unlike TFlex, TRIPS has limited options for supporting different processing granularities. For instance, TRIPS can only be configured either as an ILP engine supporting 1K in-flight instructions, or in an SMT mode with four threads each with

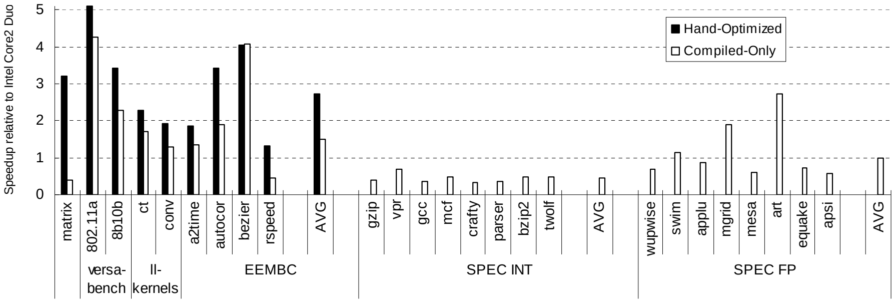


Figure 5: Relative performance (1/cycle count) for TRIPS normalized to Intel Core2 Duo

maximum of 256 instructions per thread. We only compare against the single-threaded mode of TRIPS. TFlex can instead be configured for a range of granularities to adapt to various operating targets when the need arises. Finally, having access to the TRIPS hardware design and implementation provides a solid methodology for modeling TFlex, giving us higher confidence in the performance, area, and power estimates.

Baseline Validation: For TRIPS to be a satisfactory baseline, it must achieve at least a reasonable level of performance. To establish this baseline, we compare the performance of the TRIPS hardware to that of an Intel Core2 Duo system on the suite of EEMBC, SPEC, and hand-optimized benchmarks shown in Table 1. The Intel Core2 Duo measurements were taken on a Dell E520 system that has a 2.1GHz Intel Core2 Duo processor with 2GB 533 MHz DDR2 SDRAM memory. The TRIPS system has two TRIPS processors running at 366Mhz and 2GB DDR1 SDRAM memory running at 200 MHz. The C and Fortran codes for Intel Core2 Duo were compiled using gcc 4.1.2 -O3, and the PAPI 3.5.0 library was used to collect performance counter results [5]. For the TRIPS system, performance counters also provided the precise cycle counts. All experiments use only a single core in the Core2 and TRIPS systems, and we use cycle counts as the metric for comparing performance, so that the results need not account for differences in process technology, design methodology, and size of the design team.

Figure 5 shows that on the hand-optimized benchmarks TRIPS uniformly outperforms the Core2 and achieves an average 2.7x speedup for applications run on one core of both TRIPS and Core2[1]. For the compiled benchmarks, TRIPS is approximately 50% faster on average than the Core2 on Versabench, LL kernels and EEMBC benchmarks, 3% worse on SPEC FP and 57% worse on SPEC INT. We expect higher compiler performance as the TRIPS compiler (an academic research compiler) incorporates many optimizations found in production compilers.

Simulator Validation: The simulator used in this study can model both the TRIPS hardware prototype and the TFlex microarchitecture since they both use the same ISA, and also have similar functional components such as on-chip interconnection networks, caches, execution units, and register files. The simulator was validated by simulating a configuration similar to the TRIPS prototype hardware and comparing the cycle counts against the actual hardware on a set of EEMBC benchmarks and microbenchmarks extracted from the SPEC 2000 suite. We observe that the cycle count estimates from the simulator are within 7% of the hardware cycle counts, less than the demonstrated performance gains.

6 TFlex vs. TRIPS Comparisons

Table 1 lists the microarchitectural parameters for a single TFlex core used in the experiments. The sizes of the structures in the core ensure that one core can execute and atomically commit one EDGE block. Each core has a 128-entry instruction window to accommodate all instructions in a block, 128 registers, and an LSQ large enough to hold at least 32 loads/stores. The sizes of the remaining structures, including the I-cache, D-cache, and the branch predictor, were sized to balance area overhead and performance. The simulated baseline TRIPS microarchitecture matches that described by Sankaralingam et al. [26], with the exception that the L2 capacity of the simulated TRIPS processor is 4MB to allow a fair comparison with TFlex on the multiprogramming workloads.

The TFlex architecture also includes two microarchitectural optimizations that could be applied to improve the baseline performance of the TRIPS microarchitecture. First, the bandwidth of the operand network is doubled to reduce inter-ALU contention. Second, TFlex cores support limited dual issue–two integer instructions but only one floating-point instruction issued per cycle–as opposed to the single-issue execution tiles in TRIPS.

6.1 Performance Comparison

Figure 6 shows the performance of the TRIPS prototype architecture and that of TFlex configurations ranging from 2 to 32 cores, normalized to the performance of a single TFlex core. The 26 benchmarks on the x-axis are arranged

[1] For matrix multiplication, we use the optimized binary from GotoBLAS [7, 9] for Intel Core2 Duo and we compare the FPC (FLOPS/cycle) instead of cycle counts.

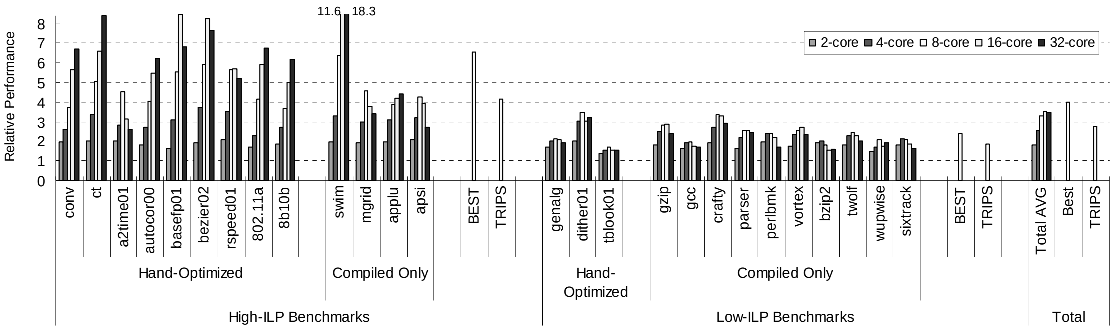


Figure 6: Performance of different applications running on 2 to 32 cores on a CLP normalized to a single TFlex core

into categories of low and high IPC. On average, the 16-core TFlex configuration performs best and shows a factor of 3.5 speedup over a single TFlex core. When the processor is configured to the best performing number of cores for each application (represented by the bar “BEST”), the performance of TFlex increases an additional 13% and the overall speedup over a single TFlex core reaches a factor of four. These results indicate that, using the proposed execution model, sequential applications can be effectively run across multiple cores to achieve substantial speedups.

On average, an eight-core TFlex processor, which has the same area and issue width as the TRIPS processor, outperforms TRIPS by 19%, reflecting the benefits of additional operand network bandwidth as well as twice the L1 cache bandwidth stemming from fine-grained distribution of the cache. Using the best per-application TFlex configuration outperforms TRIPS by 42%, demonstrating that adapting the processor granularity to the application granularity provides significant improvements in performance.

6.2 Area Comparison

We examine the area required for a TFlex processor and the performance/area as a function of the number of cores. The area of each microarchitectural component in a single TFlex core was estimated from the post-synthesis netlist (before final place-and-route) of the 130nm ASIC implementation of the TRIPS prototype. Table 2 presents the area of different microarchitectural components in an 8-core TFlex processor and a single core of a TRIPS processor. A 130nm, 18mm x 18mm die can accommodate 8 TFlex cores with 1.5MB of L2 cache. Assuming linear scaling, a 32-core TFlex array with 4MB of L2 cache would fit comfortably on a 12mm x 12mm die at 45nm.

Figure 7 plots the performance per area (1/(cycles$\times$mm^2)) for the TRIPS processor and various TFlex configurations, all normalized to a single TFlex core. For most benchmarks, area efficiency peaks either at one or two cores; beyond two cores (four-wide issue), performance improvements scale at a slower rate than area growth. Unlike conventional architectures, a composable architecture can balance area efficiency versus peak performance demand depending on various runtime factors including the number of active threads.

6.3 Power Comparison

The TFlex power model is incorporated into the TFlex simulator in a manner similar to Wattch [4], and is derived from data obtained from the TRIPS design database and hardware prototype. Because of uncertainty in scaling power models across technologies and disparate architectures, we limit the power comparisons to the 130nm technology of TRIPS and compare relative power consumption of variations of the TFlex and TRIPS microarchitectures. In the baseline TRIPS microarchitecture, we employ the models of Wattch to estimate power consumption in structures such as L1 I- and D-caches, load store queues, branch predictor tables, register files, reservation stations, ALUs, L2 caches, and on-chip network routers. We use gate and parasitic capacitances from the TRIPS design database to estimate the power consumption of the clock tree and latches. We model combinational logic power using gate and parasitic capacitance from the TRIPS netlist and simulated activity factor estimates. For leakage power, we use a simple area-based model that results in leakage of 8-10% of total power, a reasonable estimate for 130nm. The power model for an individual TFlex core consists of components from the TRIPS power model and clock tree power scaled by the ratio of TRIPS to TFlex core latch counts.

We validated the TFlex power model by configuring the multicore TFlex simulator as a single TRIPS processor and comparing the power estimates to TRIPS prototype measured power. We used the TRIPS prototype parameters of 130nm technology, 1.5V supply voltage, 366MHz processor clock frequency, and 266MHz DDR1 SDRAM frequency. On a collection of microbenchmarks used to highlight power consumption of different structures, the TFlex power model is within 10% of the measured TRIPS power. The current comparison does not account for power reduction techniques such as clock gating, as the TRIPS prototype does not implement them. However, we expect the trends and conclusions of the power efficiency of TFlex to be applicable even if power management were to be in-

Structures	8 TFlex cores (16-issue)			Single TRIPS core (16-issue)		
	Size	Area	Power	Size	Area	Power
Fetch (Block Predictor, I-cache)	66K-bit predictor, 64KB I-cache	10.9	1.66	80K-bit predictor, 80KB I-cache	7.7	0.99
Register Files	1024 entries	6.5	0.02	512 entries	3.0	0.02
Execution resources (issue window, ALUs)	1K-entry issue window 16-INT ALU, 8-FP ALU	23.6	2.24	1K-entry issue window 16-INT ALU, 16-FP ALU	39.4	2.07
L1 D-cache subsystem (D-cache, LSQ, MSHR)	64KB D-cache 352-entry LSQ	27.8	0.45	32KB D-cache 1024-entry LSQ	33.4	0.44
Routers		7.0	0.25		11.0	0.33
L2 Caches, DIMM/IO		N/A	3.34		N/A	3.34
Clock Tree, Leakage		N/A	13.52		N/A	16.95
Total		75.8	24.5		94.5	28.4

Table 2: Area (mm^2) and power estimates (*watt*) for eight-core TFlex and single-core TRIPS microarchitectures.

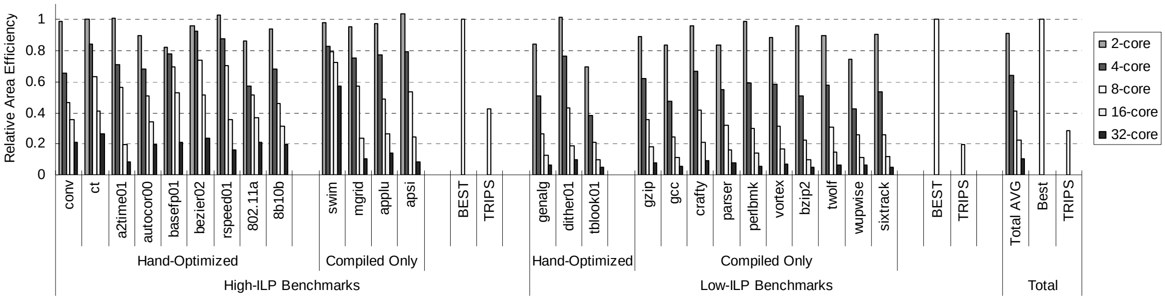


Figure 7: Performance per unit area for different applications running on 2 to 32 cores on TFlex CLP normalized to single-core TFlex.

cluded.

Power Modeling Results: Table 2 shows the average power consumed across all the benchmarks on TRIPS and an eight-core TFlex processor, as well as a breakdown of the power into categories of fetch, execution, L1 data cache, routers, L2 cache, DRAM/IO, clock tree, and leakage. The power dissipated in the individual categories is relatively small because the clock tree power in all these categories has been reported separately. Figure 8 shows power efficiency measured in performance^2/Watt for the various TFlex configurations and the TRIPS configuration, for each of the benchmarks and normalized to a single-core TFlex processor. The most power-efficient TFlex configuration ranges from four to 32, with eight cores (16-wide issue) being the best overall fixed configuration. The flexibility to choose the best composition on a per-application basis produces an average improvement of 22% over any fixed TFlex system. The power efficiency of a fixed 8-core TFlex system is about 64% better than a fixed TRIPS system. Although both have the same execution bandwidth, TRIPS has twice the number of power-hungry floating-point units, which are unused in many cycles. Fine-grained clock gating of these FPUs could improve the relative power efficiency of the TRIPS baseline.

6.4 Overhead Evaluation of Distributed Protocols

While the previous sections established the overall benefits of a scalable and composable architecture, this section examines the overheads associated with the distributed control protocols, including distributed instruction fetch/commit. When a single-threaded application runs on multiple cores, traditional architectures will require careful coordination among cores to maintain the sequential semantics of the instruction stream, especially at in-order pipeline stages such as fetch and commit. This coordination overhead can be significantly reduced if the unit of coordination is done at much larger granularity than individual instructions, such as a 128-instruction EDGE block.

Distributed Fetch: The distributed fetch protocol, including passing control from one core to another, includes six components with varying latencies, as shown in Figure 9a. The first three components incur a constant and total latency of seven cycles for a block, except for the one-core configuration which lacks speculation and thus incurs no prediction latency. Control hand-off and fetch command distribution are two latencies that both increase with the number of cores due to longer communication distances. Dispatch latency is the time to fetch from the I-cache into the instruction window, which varies depending on the number of instructions dispatched at each core.

Figure 9a shows that the overall fetch latency depends on the number of cores and is a balance between the variable overheads of control hand-off, fetch distribution, and dispatch. The largest increase comes from broadcasting the fetch command over the multi-hop network to all participating cores, which dominates when 16 or more cores are aggregated. Conversely, the effective dispatch bandwidth increases linearly with the number of cores, and the time to dispatch becomes a very small fraction of the overall latency at 16 or more cores.

Distributed Commit: Figure 9b shows the latency of

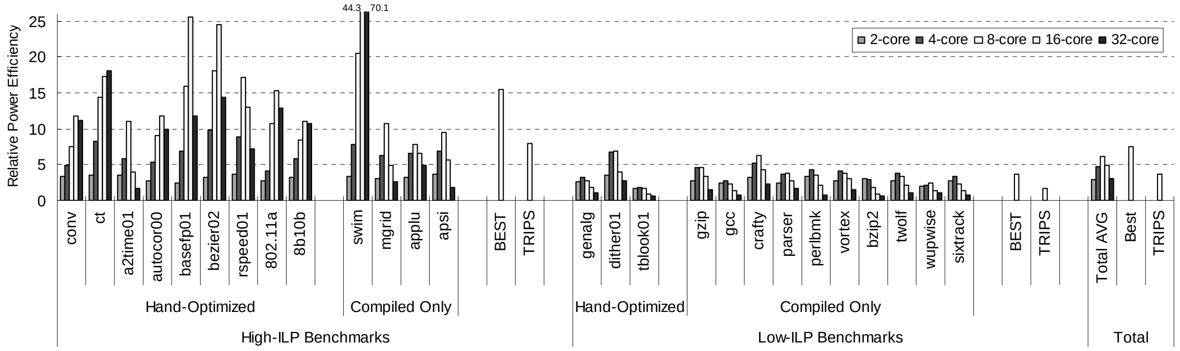


Figure 8: Performance^2/Watt normalized to a single TFlex core

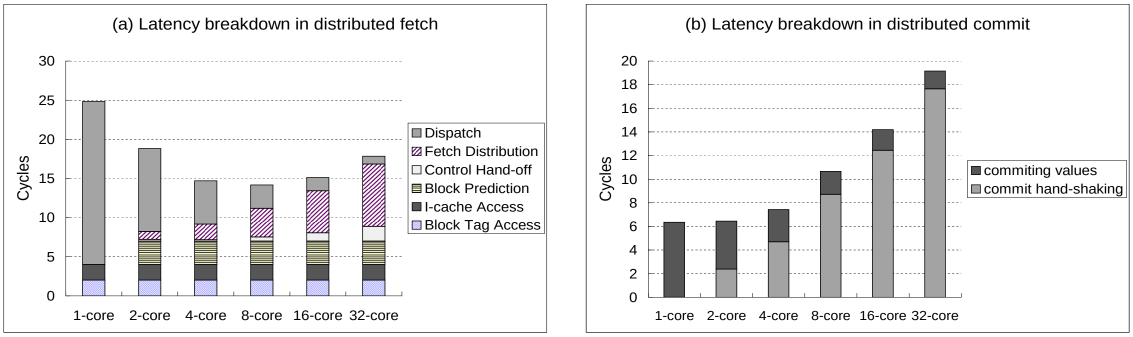


Figure 9: Coordination overheads at (a) Fetch and (b) Commit

the two principal components of commit: updating architectural state and handshaking across multiple cores. The commit handshake latency is the unique overhead to support atomic commit of blocks on a distributed execution substrate. The handshaking overhead increases with the number of cores because the control messages must travel longer distances. However, as the number of cores increases, the time to update architectural state decreases because the increase in register file and data cache bandwidth enables more of the commit to be performed concurrently.

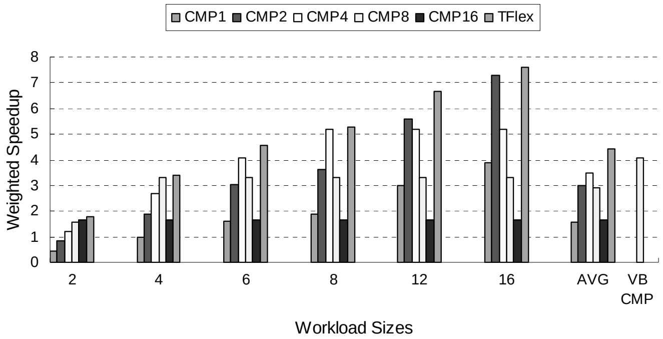
Summary: While these latencies can be significant, they will not affect performance if they are not on the critical path. To quantify the performance impact of the coordination overheads of fetch and commit, we simulated an architecture in which all of the distributed handshaking occurs instantaneously. We observed that the performance degradation was less than 2% for the largest composition (32 cores), indicating that the overheads of distributed fetch and commit can be amortized by a block-structured ISA.

7 Comparison with Chip-Multiprocessors

To ascertain the flexibility benefits of CLPs, we measured the performance of multi-programming workloads on both the TFlex design and fixed-granularity CMPs. To isolate the contributions of composability to flexibility, we modeled the fixed-granularity CMP by configuring a 32-core TFlex system to use fixed numbers of logical processors composed of equal numbers of cores (an experiment labeled “CMP-4” is a TFlex configuration that has eight composed processors with four cores each). The TFlex simulator models inter-processor contention for the shared L2 cache and main memory. The 12 benchmarks used in this experiment were selected from the hand-optimized benchmark suite, which exhibit considerable diversity in ILP. We varied the workload size (i.e. degree of multi-programming) from two to 16 applications running simultaneously.

The metric used to measure throughput improvements was *weighted speedup* (WS) [30]. For this study, we used the results from Figure 6 to compute individual benchmark speedup as a function of the number of cores, rather than employing an on-line algorithm. Given the cores-to-speedup function of each application, we use an optimal dynamic programming algorithm to find the core assignments that maximize WS on TFlex [11]. Each workload was run only on those fixed CMPs that had at least as many processors as applications in the workload. When the workload size exceeds the CMP processor count, we assume that the weighted speedup stays constant, similar to a previously applied methodology [19].

Results: For each workload size, Figure 10 reports the average weighted speedup on TFlex and on the set of fixed-granularity CMPs. The best CMP granularity changes depending on the workload size (i.e. the number of threads). CMP-16 is the best for workloads with two threads, CMP-8 is the best for four-thread workloads, CMP-4 is the best



Workload Sizes	1-core	2-core	4-core	8-core	16-core	32-core
2	0%	0%	13%	35%	53%	0%
4	0%	0%	31%	54%	15%	0%
6	0%	18%	43%	37%	2%	0%
8	0%	28%	59%	14%	0%	0%
12	9%	82%	9%	0%	0%	0%
16	32%	52%	16%	0%	0%	0%

The percentage of applications mapped to each composed processor in TFlex for each workload size

Figure 10: Weighted Speedup comparison between TFlex and fixed-granularity CMPs.

for six to eight threads, and CMP-2 is the best for twelve to sixteen threads. Because of the capability of selecting the best granularity for each workload, the TFlex array consistently outperforms the best fixed-granularity CMP. The set of bars labeled AVG show the average WS across all workloads. While CMP-4 is the best granularity for fixed CMPs, the TFlex design produces an average of 26% higher WS and a maximum of 47% higher WS.

Figure 10 also demonstrates the value of asymmetric processor composition in TFlex by comparing to a hypothetical flexible CMP that can change its granularity dynamically but is symmetric, requiring each processor to be composed from equal numbers of cores. The results for this hypothetical system, labeled VB CMP (Variable Best CMP), show that the capability of composing processors of different granularities for different simultaneously executing programs results in a 6% improvement in throughput. For a given granularity, even when the number of threads exactly matches the number of processors, some threads may under-utilize their own processor's resources. In such a scenario, TFlex is able to allocate a portion of these under-utilized resources to applications which can make better use of them. The table in Figure 10 lists the fraction of applications at each workload size that are assigned to a given processor granularity and demonstrates that the optimal processor granularity varies even within a given workload size. For workloads of size eight, TFlex allocates four cores to the majority of the applications, but also allocates two cores to 28% of them and eight cores to 14% of them. A CMP-4 configuration, conversely, would allocate only four cores regardless of the application characteristics.

8 Conclusions

Since clock rate scaling can no longer sustain computer system performance scaling, future systems will need to mine concurrency at multiple levels of granularity. Depending on the workload mix and number of available threads, the balance between instruction-level and thread-level parallelism will likely change frequently for general-purpose systems, rendering the design-time freezing of processor granularity in traditional CMPs an undesirable option. Composable lightweight processors (CLPs) provide the flexibility to allocate resources dynamically to different types of concurrency, ranging from running a single thread on a logical processor composed of many distributed cores to running many threads on separate physical cores. The system can also use energy and/or area efficiency to choose the best-suited processor configuration. We envision multiple methods of controlling the allocation of cores to threads. At one end of the spectrum, the operating system could make such decisions based on the number of threads to run, and their criticality. The OS could even monitor how each thread uses its allocated resources and reallocate them among the threads as necessary. At the other end of the spectrum, the hardware could potentially adjust the number of cores per thread in an automated fashion. This capability would further blur the distinction between conventional uniprocessors and multiprocessors, which we view as a promising direction.

9 Acknowledgments

We thank Ramdass Nagarajan, Haiming Liu, Mark Gebhart, Bert Maher, Katherine Coons, Jeff Diamond and Behnam Robatmilli for their contribution to the paper. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

References

[1] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 248–261, 1999.

[2] D. H. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(12):49–58, 2003.

[3] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 218–229, 2001.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[5] S. Browne, J. Dongarra, N. Garner, K. S. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 conference on Supercomputing*, page 42, 2000.

[6] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7), 2004.

[7] J. Diamond, B. Robatmilli, S. W. Kecker, R. van de Geign, K. Goto, and D. Burger. High performance linear algebra on a spatially distributed processor. Technical Report TR-07-49, Department of Computer Sciences, The University of Texas at Austin, 2007.

[8] D. Folegnani and A. González. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 230–239, 2001.

[9] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.

[10] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. K. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2), 2000.

[11] T. Ibaraki and N. Katoh. *Resource allocation problems: Algorithmic approaches*. MIT Press, 1988.

[12] E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 186–197, 2007.

[13] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[14] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2), 2005.

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

[17] 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 *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, 2003.

[18] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 195–206, 2004.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 64–75, 2004.

[20] F. Latorre, J. González, and A. González. Back-end assignment schemes for clustered multithreaded processors. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 316–325, 2004.

[21] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, pages 105–118, 2005.

[22] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 90–101, 2001.

[23] R. M. Rabbah, I. Bratt, K. Asanović, and A. Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Massachusetts Institute of Technology Technical Report MIT-LCS-TM-646, 2004.

[24] P. Racunas and Y. N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 22–31, 2003.

[25] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 422–433, 2003.

[26] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 480–491, 2006.

[27] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-Binding: Enabling unordered load-store queues. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 347–357, 2007.

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

[29] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4-5):505–522, 2005.

[30] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[31] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

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

[33] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[34] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[35] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the International Conference on High Performance Computer Architecture*, pages 25–36, 2007.