Strategies for Mapping Dataflow Blocks to Distributed Hardware

Behnam Robatmili, Katherine E. Coons
Department of Computer Sciences
The University of Texas at Austin
{beroy,coonske}@cs.utexas.edu

Doug Burger
Microsoft Research
One Microsoft Way Redmond, WA 98052
dburger@microsoft.com

Kathryn S. McKinley
Department of Computer Sciences
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Distributed processors must balance communication and concurrency. When dividing instructions among the processors, key factors are the available concurrency, criticality of dependence chains, and communication penalties. The amount of concurrency determines the importance of the other factors: if concurrency is high, wider distribution of instructions is likelier to tolerate the increased operand routing latencies. If concurrency is low, dependent instructions should be mapped close to one another because communication costs are likely to contribute to the critical path.

This paper explores these tradeoffs for distributed Explicit Dataflow Graph Execution (EDGE) architectures that execute blocks of dataflow instructions atomically. We introduce a runtime block mapper *that assigns instructions from a single thread to distributed hardware resources (cores) based on compiler-assigned instruction identifiers. Two classes of approaches are possible for mapping instructions:* fixed *strategies that map all blocks to the same number of cores, and* adaptive *strategies that vary the number of cores for each block. The results show that best fixed strategy varies, based on the cores' issue width. A simple adaptive strategy improves performance over the best fixed strategies for single and dual-issue cores, but its benefits decrease as the cores' issue width increases. These results show that by choosing an appropriate runtime block mapping strategy based on the microarchitecture of the cores, average performance can be increased by 18%, while simultaneously reducing average operand communication significantly, by 70%, saving energy as well as improving performance. These results indicate that runtime block mapping is a promising mechanism for balancing communication and concurrency in distributed processors.*

1. Introduction

Balancing concurrency and communication is a fundamental challenge when mapping instructions to a distributed substrate. As the granularity of parallel computation increases the frequency and cost of communication changes, as does the extent to which the parallel units of computation can be statically summarized. We investigate the tradeoff between communication and concurrency for the case where the parallel unit of computation is a fixed-size block of instructions. Instruction-level parallelism can be exploited by mapping each block of instructions to multiple cores. Block-level parallelism can be exploited by mapping multiple blocks of instructions to the substrate at the same time.

We introduce a run-time block mapper, implemented in hardware, that maps these blocks to a distributed substrate of composable cores. The block mapper can use various policies to map blocks to cores that represent different tradeoffs between concurrency and communication. The evaluation platform for this block mapper is the TFlex microarchitecture, a composable lightweight processor that executes blocks of instructions atomically on a distributed substrate [1]. TFlex implements an EDGE ISA, in which the instructions within an atomic block encode their targets and communicate with one another directly. Instructions encode their targets using a 7-bit identifier (ID) associated with each instruction. The compiler encodes criticality and locality information in the ISA when it assigns these IDs. Then, the block mapper assigns the instructions in each block to cores based on their IDs and the number of available resources as provided by the operating system.

We first consider a spectrum of *fixed* policies in which the block mapper maps all blocks to a fixed number of cores. At one extreme, a *flat* strategy spreads the instructions within a block across all participating cores. This flat strategy achieves high performance with single-issue cores, at the cost of frequent operand communication. At the other extreme, a *deep* strategy maps all of the instructions in a block to only one core. This strategy performs well for dual-issue cores, which are able to exploit intra-block parallelism locally while reducing operand traffic significantly.

We also explore an *adaptive* strategy, in which the block mapper uses a compiler specified concurrency value to adjust the number of cores to the block. Results show that adaptive outperforms fixed strategies on both single and dual-issue cores. When running on single-issue cores, the adaptive strategy achieves higher performance than the flat strategy with operand traffic comparable to that of the deep mapping strategy. Our results also suggest that if a future compiler can effectively express more information about communication and criticality, it is likely to inform block mapping policies that further improve performance.

2. Related Work

To support workloads with differing degrees of parallelism, multi-core systems must adapt the granularity of cores to match the available number of threads [2]. One approach to this problem is to aggregate a small number of cores to form a larger core capable of exploiting concurrency at a finer granularity [3], [1]. Recent studies propose methods for aggregating both in-order [4], [5] and out-of-order cores [3], [1]. This study relies on out-of-order core aggregation as the underlying mechanism for exploiting block-level concurrency in programs.

Some architectures, such as VLIW architectures and RAW, rely heavily on the compiler to map instructions to a distributed substrate. For example, the RAW compiler schedules instructions in time to exploit concurrency, and places instructions on a physical substrate [6]. The Voltron architecture [4] combines multiple in-order VLIW cores into a wide-issue VLIW core. This statically exposed architecture relies on the compiler to schedule VLIW instructions and extract fine-grained communicating threads.

Fully dynamic approaches only use hardware to map instructions. These methods do not take advantage of instruction dependencies extracted by the compiler. Clustered superscalar processors [7], [8] rely on the hardware to steer instructions dynamically to different clusters based on instruction dependencies. Complexity-Effective Superscalar Processors steer the dependent instructions into separate FIFO buffers dynamically and only send the result tags to the heads of the FIFO buffers [9]. The ISA for Instruction Level Distributed Processing [10] supports hierarchical register files consisting of many general purpose registers and a few accumulator registers. The instruction stream is divided into short strands of dependent chains. The instructions in each strand are steered into a processing element associated with the accumulator accessed by those instructions. While the instructions in each cluster are linked by the the accumulator, the inter-strand dependencies are passed through the general purpose registers. To simplify the hardware, this paper relies on the compiler to specify instruction dependencies and concurrency, rather than discovering it at runtime.

The runtime mapping approach presented here, which can use static information, is most similar to approaches in which the hardware allocates/maps coarse chunks of work to distributed units, often with compiler support. The compiler for Multicluster processors partitions instructions between clusters during register allocation to minimize remote register accesses [11]. Instructions in each cluster are scheduled dynamically by the hardware. In Multiscalar [12] and Thread-Level Speculation [13], the hardware automatically spawns speculative threads, selected by the compiler, on multiple cores. These more speculative approaches rely on discontiguous instruction windows.

A recent trend has been to balance ILP and TLP by adjusting the number of distributed resources allocated to a thread dynamically, by having multiple independent units collude to accelerate a single thread. This approach makes distribution of instructions more challenging because the number of participating processor elements is unknown statically and may change dynamically. In the Federation technique [5], two neighboring in-order cores, similar to Niagara/T1 [14] cores, are "federated" to create an out-of-order processor. A recent study, however, demonstrates that aggregating in-order cores, even under idealized assumptions about aggregation overheads, leads to major performance challenges [15]. Some recent work has allowed core aggregation on a set of out-of-order cores. CoreFusion [3] is a technique that "fuses" multiple dual-issue out-of-order cores to form a wide-issue out-of-order core. The fused cores form a distributed instruction cache, instruction window and branch predictor, but some of the structures, such as register renaming, are physically shared, which limits the aggregate issue width to eight.

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 is an early EDGE design that uses the compiler to form predicated *blocks* of dataflow instructions and to specify the placement of each instruction on a grid of 16 ALUs, where they are issued dynamically [16]. Wavescalar is a dataflow processor that uses static placement of instructions and dynamic issue on a hierarchical substrate [17]. Neither design explicitly supports dynamic variance of the available hardware resources for mapping. TFlex is a second generation EDGE design that supports dynamic core aggregation [1], and is the underlying distributed substrate used in this paper.

3. System Overview

TFlex is a composable lightweight processor in which all microarchitectural structures, including the register file, instruction window, predictors, and L1 caches are distributed across a set of cores [1]. Distributed protocols implement instruction fetch, execute, commit, and misprediction recovery on this distributed substrate without centralized logic. To run a program, the OS assigns a set of cores on the substrate to a program, and the program treats those cores as a single

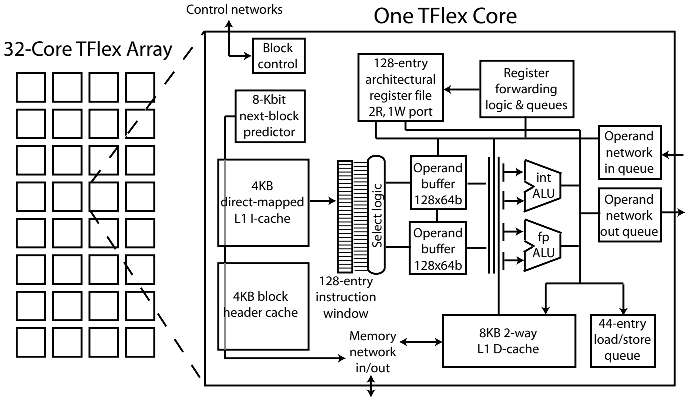


Figure 1. Microarchitectural components of one core of a 32-core TFlex Composable Light-weight Processor.

processor. Figure 1 illustrates the various microarchitectural components of a TFlex core [1].

When cores are aggregated, the register file, instruction cache, and data cache are equally distributed across all participating cores. Each operation that needs to access a microarchitectural structure uses a hash function to determine on which core the structure to be accessed resides. For example, the low-order bits of the cache index select the core on which the data cache bank to be accessed resides, and the low-order bits of the architectural register number select the core holding that register in its register file [1].

TFlex implements an EDGE ISA that supports block-atomic execution, so fetch, completion, and commit protocols operate on blocks rather than individual instructions. Each instruction within a block explicitly encodes its consumer instructions, which eliminates the need for an operand broadcast network. Instead, a point-to-point network between cores performs producer-consumer communication using instruction IDs.

Figure 2 shows the components in a TFlex system. The compiler [18] breaks the program into single-entry, predicated blocks of instructions, similar to hyperblocks [19]. The EDGE ISA imposes several restrictions on these blocks. The maximum size of each block is 128 instructions, and each block can contain up to 32 register reads, 32 register writes, and 32 load/store instructions. During compilation, the instruction scheduler assigns an identifier to each instruction.

The scheduler takes as input the hardware topology, which includes the maximum number of participating cores, network latencies, and the mapping of instruction identifiers to cores. The scheduler seeks an assignment of IDs that minimizes the latency of the critical path through the block by minimizing communication along that path and exploiting available concurrency [20]. The 7-bit ID associated with each instruction is encoded implicitly in the binary via the location of the instruction within the block. The block mapper uses these instruction IDs to map instructions to the distributed substrate at runtime. The instruction scheduler and block mapper agree upon an interface such that these IDs can be used to convey static information about locality and criticality. The block mapper uses this locality and criticality information to determine on which participating core it will place the instruction and in what order that instruction will appear in the reservation station.

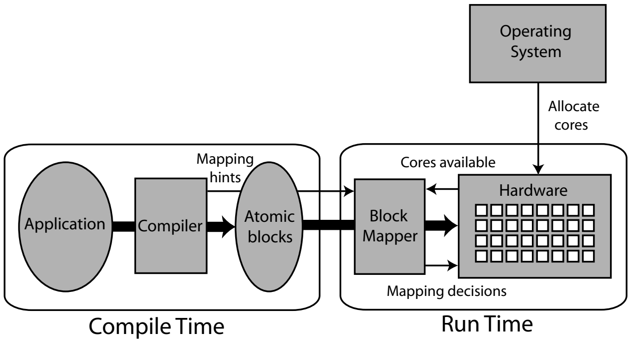


Figure 2. Runtime and compile time system components

The runtime system allocates N cores to an application based on resource availability. The hardware fetches and executes up to N blocks in parallel on the N participating cores, where N is a power of 2. One executing block is always non-speculative and the others are speculative. The mapping determines how many instructions from the same block a core executes. For example, a core can execute up to 128 instructions from the same block, or $128/N$ instructions from N different blocks. Inter-block communication occurs via registers, cache, and memory based on the hash functions. Intra-block communication between instructions depends on the dataflow graph, the number of participating cores, and the mapping of blocks to participating cores.

At runtime, a block mapper maps the instructions in each block to the participating cores using a block mapping policy and the IDs provided by the compiler. The cores selected for running a block fetch and execute the instructions inside the block.

4. Block Mapping Strategies

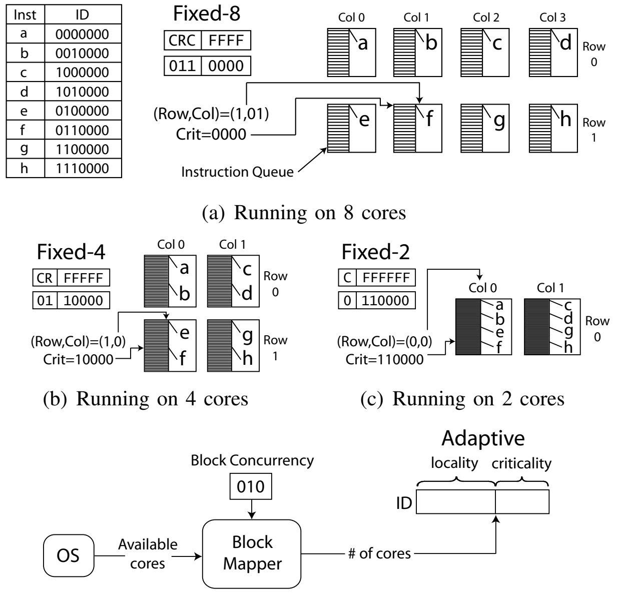
For a given block, the block mapper may choose to distribute the block across all participating cores, a subset of these cores, or a single core. Each strategy represents a different tradeoff between parallelism and communication overhead. We explore *fixed* and *adaptive* strategies. The fixed mapping strategies choose the same number of cores for all blocks in a program. At one extreme, the fixed *flat* strategy partitions the block across all participating cores, exploiting intra-block concurrency. At the other extreme, the fixed *deep* strategy puts the entire block on a single core, minimizing intra-block communication. The adaptive strategy seeks a better tradeoff by choosing the number of cores based on block characteristics. For each of these block mapping strategies, the block mapper interprets IDs assigned to each instruction by the compiler. We next describe this software/hardware contract in more detail, and then fully discuss each mapping strategy.

4.1. Contract between the Compiler and Hardware

We use IDs to express criticality and locality. The block mapper reinterprets these IDs to allow programs to run on a different number of cores without being recompiled. Because there are at most 128 instructions in a block, the compiler assigns each instruction a 7-bit ID that determines *where* the instruction will execute, i.e., on which core. At runtime, instructions execute *when* their operands arrive. If two instructions on the same core are both ready at the same time, the more critical instruction should execute first. The block mapper uses the IDs to determine the order in which instructions appear in the reservation stations on each core, thus, the ID can be used to express criticality information as well as locality information.

The instruction IDs should preserve locality information if the block is mapped to a smaller number of cores. As a simple example, consider 32 cores laid out in a 4 by 8 grid. The compiler and hardware could agree that IDs 0-3 map to core (0,0), 4-7 to (0,1), and so on. At runtime, if there are only four participating cores laid out in a 2 by 2 grid, the block mapper must interpret the bits differently, for instance by mapping IDs 0-31 to (0,0). The problem with this simple mapping is that instructions that were one hop away, those mapped to (2,4) and (3,4) in the 4 by 8 grid, are now assigned to (0,1) and (1,0), which are two hops away in the 2 by 2 grid. Ideally, the IDs should be assigned and interpreted such that instructions mapped to the same or nearby cores when compiled for N cores remain on the same or nearby cores when mapped to a smaller number of cores. We use the following encoding to achieve this versatility.

Figures 3(a-c) show the hardware/software contract for ID bits when running on eight, four, and two cores, respectively. With eight cores, each core will execute 16 of the 128 instructions and the first three bits determine the core. The scheduler encodes locality information in these top three bits: R (row) and C (column) in the figures. The four remaining frame (F) bits express criticality information, where lower is more critical and appears earlier in the reservation stations. The core chooses to execute the lower numbered instructions when two instructions are ready to issue in the same cycle. Similarly, mapping to four and two cores, the microarchitecture uses two and one locality bits, and five and six criticality bits, respectively.



(a) Running on 8 cores

(b) Running on 4 cores

(c) Running on 2 cores

(d) Adaptive mapping

Figure 3. Information encoded in the instruction IDs for fixed and adaptive mapping strategies.

By interleaving the R and C bits in the instruction IDs, the compiler helps the hardware preserve locality information when mapping blocks to different numbers of cores. For example, in Figure 3(a), the scheduler maps dependent instructions a and b to two adjacent cores, and independent instructions a and h to two distant cores. At runtime, when mapped to four and two cores as shown in Figures 3(b) and 3(c), the relative locality among these instructions is preserved. This format for IDs, however, does not preserve the criticality of instructions because as instructions are mapped to fewer cores, locality bits are converted to criticality bits. For example, all eight instructions in Figure 3(a) have high criticality and are thus placed in the highest position in their cores' reservation stations. When mapped to four and two cores, as shown in Figures 3(b) and 3(c), however, the relative positions of these instructions in their reservation stations change dramatically. Fortunately, instructions are allowed to issue out of order, so the criticality bits only become a factor when multiple instructions are ready to execute at the same time.

4.2. Fixed Mapping Strategies

Each fixed strategy represents a different tradeoff between communication overhead and ability to exploit concurrency.

4.2.1. Flat Mapping. With a flat mapping strategy, the block mapper distributes the instructions in each block across all participating cores. This approach exploits as much intra-block concurrency as possible, but incurs high intra-block communication overheads.

The IDs convey both locality and criticality information with the flat mapping strategy. For example, in a 4×4 configuration containing 16 total cores, each of the 16 cores execute 8 of the 128 instructions as shown in Figure 3. The flat mapper uses four bits to indicate the location of the instruction, two bits for the row, and two bits for the column. The remaining three bits express criticality information – the relative issue priority that breaks ties in the reservation stations on each core (see Figure 3). Instructions that are close to each other when compiled to 16 cores are close, or on the same core, when executed in a flat mapping on a smaller number of cores. The TRIPS prototype employed what was essentially a flat mapping strategy across 16 execution tiles [16].

4.2.2. Deep mapping. With a deep mapping strategy, the block mapper assigns all instructions within a block to a single core. This strategy eliminates cross-core communication between instructions, but provides only as much intra-block parallelism as the issue width of the cores. Although deep mapping eliminates communication between instructions, it may increase communication between blocks because cache banks and registers are distributed across the cores.

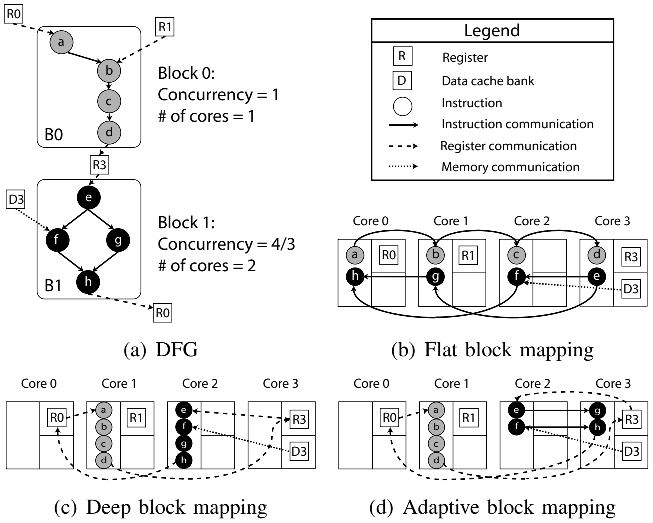
With the deep mapping strategy, the instruction identifiers assigned by the scheduler are no longer used for locality at all – the entire instruction identifier is devoted to determining the criticality of the instruction, i.e., the instruction's priority within the core's reservation stations.

For the DFG in Figure 4(a), Figures 4(b) and 4(c) provide a simple example of the flat and deep mapping strategies for two blocks, $B0$ and $B1$, on a 4-core processor. Symbols a through h represent the instructions in these blocks. Registers $R0$, $R1$, and $R3$ are located in cores 0, 1, and 3, respectively. Block $B0$ reads registers $R0$ and $R1$, and writes register $R3$. Block $B1$ reads register $R3$, which is produced by $B0$, and writes register $R0$. Block $B1$ also loads a value from cache bank $D3$ located on core 3.

The value communicated between blocks $B0$ and $B1$ via register $R3$ is an example of communication between blocks, while the value produced by instruction a and consumed by instruction b is an example of communication within a block. With flat mapping, as shown in Figure 4(b), the instruction scheduler tries to place instructions that access registers on the same core as the corresponding register. With deep mapping, as shown in Figure 4(c), however, the blocks are assigned to cores dynamically in a round-robin fashion, so most register accesses go to remote cores.

4.3. Adaptive Mapping

Flat and deep mapping are both limited because the block mapper selects the same number of cores, C, for all blocks in an application. The flat mapping strategy uses $C = N$, where N is the number of participating cores. The deep mapping uses $C = 1$. As a result, flat mapping may under-utilize cores or experience excessive communication overheads when blocks have low concurrency. On the other hand, the deep mapping fails to exploit all of the available concurrency for highly concurrent blocks.



(a) DFG

(b) Flat block mapping

(c) Deep block mapping

(d) Adaptive block mapping

Figure 4. A sample DFG consisting of two blocks mapped using the flat, deep and adaptive mapping strategies. Solid and dotted lines represent intra and inter-block communication, respectively.

The adaptive mapping strategy balances these tradeoffs by selecting the number of cores based on the block's available concurrency and then using the IDs to map to the selected cores. The compiler evaluates the available concurrency and encodes the concurrency value in the block header as follows:

$$Concurrency = \frac{BlockInstructionCount}{CriticalPathLength}$$

where *BlockInstructionCount* is the total number of instructions in the block and *CriticalPathLength* is the length of the critical path through the block in cycles. This metric estimates the maximum achievable IPC for the block. At runtime, the block mapper dynamically selects a set of cores for the block based on the concurrency value provided by the compiler as follows:

$$C = 2^{\lceil \log_2 \lceil \frac{Concurrency}{IssueWidth} \rceil \rceil}$$

where *IssueWidth* is the issue width of each core, assuming homogeneous cores. The block mapper uses this number of cores, always a power of two, if they are available.

Using the adaptive strategy, a round-robin algorithm chooses the cores for the next block, similar to deep mapping, but it also accounts for requests with varying numbers of cores. If there is not enough room in the instruction window for the next block, then instruction fetch stalls until there is sufficient space available. More sophisticated algorithms are possible, but may make the hardware implementation impractical. Round-robin strategies can be implemented in a distributed fashion without any centralized components.

Figure 4(a) shows the concurrency and core count for blocks $B0$ and $B1$, and Figure 4(d) illustrates the adaptive block mapping for these blocks on a 4-core processor. For simplicity, this example assumes that the static execution time for all instructions is one cycle, and that all cores have an issue width of one. $B0$ consists of a chain of dependent instructions, and all of its instructions are on its critical path. As a result, its concurrency is equal to 1.0, and the block mapper assigns one core to this block. On the other hand, the length of the critical path of block $B1$ is three cycles, but this block has four instructions, which results in a concurrency value of 4/3. For this block, the number of cores chosen by the block mapper is equal to $2^{\lceil \log_2 \lceil \frac{4/3}{1} \rceil \rceil} = 2^{\lceil \log_2 2 \rceil} = 2$. If the cores were dual-issue, the concurrency values for $B0$ and $B1$ would remain the same, but the block mapper would assign one core to each of the blocks in this example.

As shown in Figure 3(d), the adaptive strategy uses the concurrency information for each block to select an appropriate number of cores for that block. At runtime, this number determines how many bits in the instruction identifier specify locality and how many bits specify criticality.

4.4. Reducing Communication Between Blocks

One disadvantage of the deep and adaptive block mapping strategies is that they may increase communication between blocks. One way to deal with this problem is to use a different algorithm to select the next core in the block mapper. We investigate two possible algorithms.

Inside-Out. The Inside-Out algorithm prioritizes the cores close to the center when selecting the next set of cores at runtime. Because the cores close to the center have a smaller average distance to other cores, they should require a smaller average hop count to access registers and memory locations.

Preferred-Location. The compiler encodes a list of preferred cores in the block header. During core selection, the Preferred-Location block mapper selects the available cores highest in the list. To prioritize the cores, the compiler computes the static hop count required to access registers. For example in Figure 4(d), block *B0* prefers core 1 to core 0 because core 1 will require two cycles of latency to read *R0* and *R1*, and write *R3*, whereas core 0 will require three cycles. If cores 0 and 1 are both available at runtime for *B0*, the block mapper chooses core 1. A drawback of this algorithm is that the compiler must know the number of cores assigned to the program, which makes it less general than Inside-Out selection.

4.5. Hardware Complexity and Cost

The dynamic block mapper for deep and adaptive strategies can be implemented in a fully distributed way among cores, thus, there is no central unit for making block mapping decisions. Distributing the block mapper among cores minimizes its effect on the latency of the critical path. Here, we briefly discuss various components in this distributed block mapper.

Next core selection mechanism. The core selection mechanisms can be implemented in a fully distributed fashion. For the deep mapping strategy, the selected core for the current block sends a message to the next core in round-robin order to execute the next block. This mechanism requires no extra state in the cores. The adaptive block mapping strategy, however, requires each core to keep track of the allocation status of other cores in a table consisting of $N * log_2 N$ flip flops, where N is the total number of cores. In addition, each core requires a priority encoder to choose the next set of cores using the table. The table and encoder incur a relatively small area overhead for each core.

Decoding IDs. The block mapper specifies how each core interprets IDs. For example in the deep strategy, all seven ID bits determine the position of each instruction in the core's reservation stations. In the flat strategy, the mapper uses $7 - log_2 N$ bits as criticality bits. In the adaptive strategy, C cores use $7 - log_2 C$ bits for criticality.

5. Results

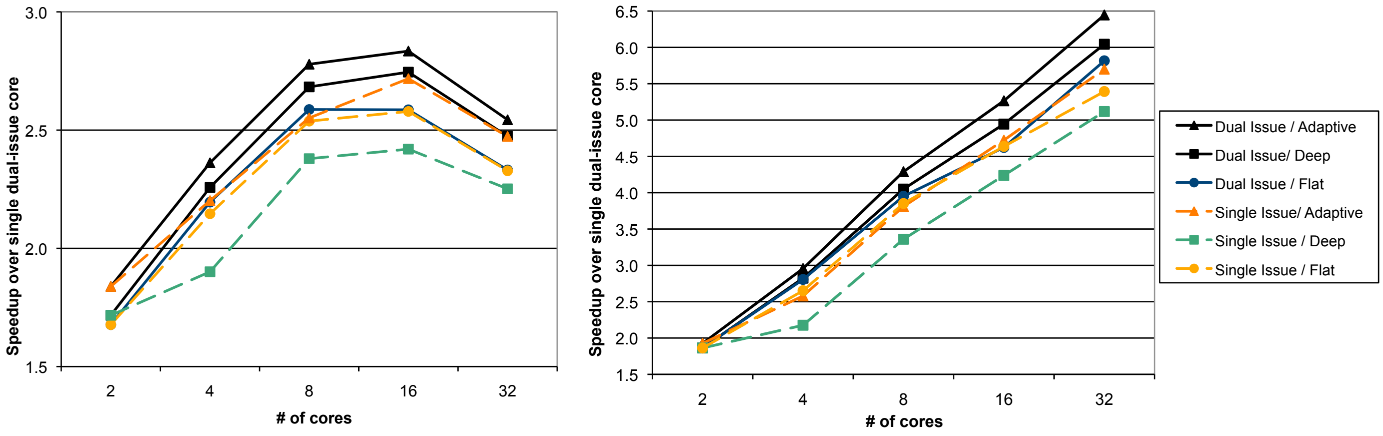
We added support for the fixed and adaptive strategies to the validated TFlex simulator [1]. Throughout this section, unless otherwise specified, we report speedup over the corresponding single core configuration. The baseline cores are capable, dual-issue, out-of-order cores with a 128-instruction window. Table 1 shows the microarchitectural parameters of each single TFlex core.

We test each mapping strategy on the SPEC [22] benchmarks. We use nine integer and 10 floating point SPEC benchmarks with the reference (large) dataset simulated with single SimPoints [23].

Throughout this section, unless otherwise specified, we run the flat strategy with binaries scheduled for 32 cores, the largest grid size. We compile to the maximum number of cores because in a real system, the number of cores assigned to a program may not be known at compile time. Binaries compiled to a smaller number of cores may sacrifice locality information when running on a larger number of cores, which can lead to a large performance degradation. The deep strategy uses binaries scheduled specifically for one core such that all seven bits in the instruction identifiers are used for criticality. The Fixed-2 and Fixed-4 strategies use binaries sheduled for two and four cores, respectively. The adaptive strategy uses binaries scheduled for eight cores because few

Table 1. Single Core TFlex Microarchitecture Parameters [1]

Parameter	Configuration
Instruction Supply	Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Num. entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP) or single issue.
Data Supply	Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 44-entry LSQ bank; 4MB decoupled S-NUCA L2 cache [21] (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



(a) SPEC INT

(b) SPEC FP

Figure 5. Average speedup over single core for the SPEC benchmarks with varying numbers of cores and varying core issue widths.

blocks require more than eight cores with this strategy. This section includes experimental results for different strategies, the concurrency distribution among blocks for the adaptive strategy, the communication overhead of each strategy, and mechanisms for improving the inter-block communication in the deep and adaptive strategies.

5.1. Performance

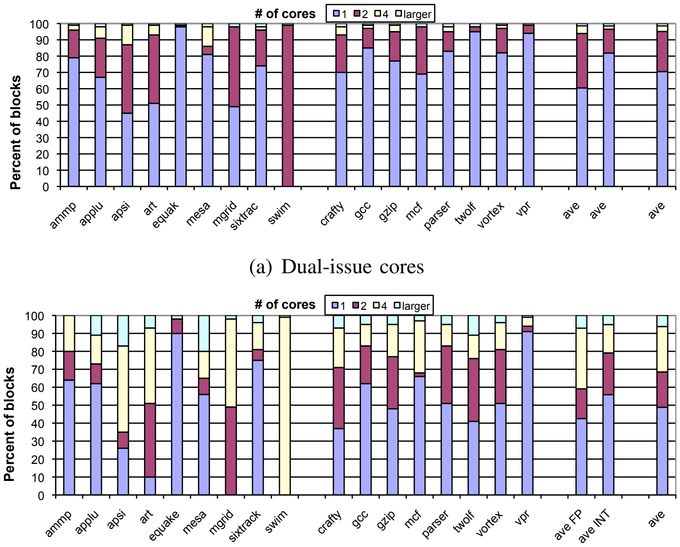
Figure 5 shows performance using the flat, deep, and adaptive mapping strategies for the SPEC benchmarks normalized to the performance of each benchmark on a single dual-issue core. These experiments vary the number of cores allocated to the application from 1 to 32 cores, and the issue width of the cores from one to two. The baseline cores, however, are always out-of-order, dual-issue cores.

With dual-issue cores, the adaptive strategy outperforms the fixed strategies in all cases. With the flat mapping strategy, there is little benefit in moving from single to dual issue cores. Both the deep and the adaptive strategies, however, see a noticeable improvement moving from single to dual-issue cores. This difference is more pronounced for the deep mapping strategy because with single-issue cores the deep strategy is unable to exploit any concurrency within a block. The adaptive strategy, however, is able to compensate for the loss in intra-block parallelism by using more cores when the block contains sufficient concurrency.

In addition to the deep and flat strategies, we also measured speedup using two intermediate fixed stratrgies. With the Fixed-2 and Fixed-4 strategies, the scheduler schedules all blocks to two and four cores, respectively, and the block mapper selects two and four cores for each block in a round-robin fashion. In most configurations, the adaptive strategy achieves performance close to or better than the performance of the best fixed strategy. In single issue SPEC FP runs, the Fixed-4 strategy achieves better performance than the adaptive strategy when running on 8 and 16 cores.

Figure 6 indicates the percentage of executed blocks that use each number of cores with the adaptive mapping strategy for the SPEC benchmarks. With dual-issue cores, as shown in Figure 6(a), the adaptive strategy maps about 40% of blocks to two or four cores. The block mapper maps 20% of the blocks in the SPEC integer becnhmarks to more than one core when using dual-issue cores. When using single-issue cores, as shown in Figure 6(b), half as many blocks are mapped to more than one core, and more than half of the blocks use two or four cores.

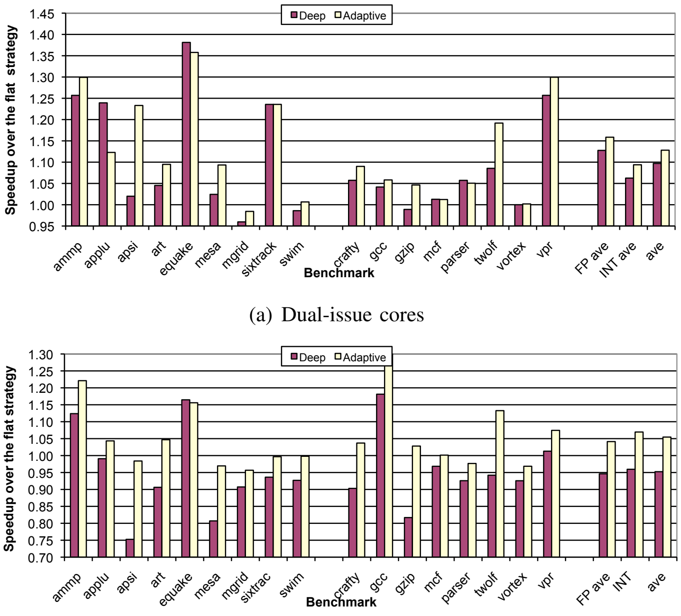
The SPEC integer benchmarks reach their maximum performance when running on 16 cores but observe a significant slowdown with 32 cores. High operand network latency and contention due to register and memory traffic when using 32 cores is the most likely reason for this slowdown. The



(a) Dual-issue cores

(b) Single-issue cores

Figure 6. Percent of blocks mapped to each number of cores by the block mapper for SPEC benchmarks.



(a) Dual-issue cores

(b) Single-issue cores

Figure 7. Speedup over the flat mapping strategy for the SPEC benchmarks with 16 cores.

adaptive block mapping strategy reduces this degradation to some extent, but cannot remove the inter-block communication. The performance of the SPEC floating point benchmarks improves when running with 32 cores, however, most likely because the floating point benchmarks contain blocks that can actually exploit 32 cores of parallelism.

Figure 7(a) shows the speedup achieved using the adaptive and deep mapping strategies normalized to the flat mapping strategy for individual SPEC benchmarks on 16 dual-issue cores. For most programs, the deep and adaptive mapping strategies outperform the flat mapping strategy. Some benchmarks, including *ammp*, *equake*, *sixtrack*, and *vpr*, show significant speedups when using the deep and adaptive mapping strategies because the most critical blocks in these applications have little concurrency available. For instance, the benchmark with the largest speedup using the deep and adaptive mapping strategies is *equake*, and the adaptive strategy for *equake* chooses to place 98% of dynamically executed blocks on a single core. This result indicates that the instruction scheduler is able to find very little concurrency in the most critical blocks of this benchmark, so the flat mapping strategy incurs extra communication overhead without any benefit in parallelism. Similarly, the mapper chooses to map 94%, 79%, and 74% of blocks to only one core in *vpr*, *ammp*, and *sixtrack*, respectively, as shown in Figure 6(a).

For most SPEC benchmarks, the adaptive mapping strategy performs better than the deep mapping strategy. The *apsi* benchmark achieves the largest speedup using the adaptive strategy, with a speedup of about 1.2. For this benchmark, the block mapper chooses to map about 50% of blocks to two or four cores as shown in Figure 6(a). This mapping suggests that there is a high amount of concurrency available in this benchmark, and the adaptive mapping strategy is able to exploit it.

Figure 7(b) graphs the performance of individual SPEC benchmarks when running on single-issue cores using the flat, deep, and adaptive strategies. For most benchmarks, the deep strategy performs worse than the flat strategy because there is more intra-block concurrency than the single core can support. The adaptive strategy outperforms both the deep and and the flat strategy for most benchmarks.

5.2. The Effect of Cross-Core Communication on Performance

To find the extent to which the overhead of inter and intra-block communication affects performance, we experimented with the following idealized modes:

- **Baseline**: The results discussed in the previous section.
- **Perfect-Reg**: In this mode, accessing a register on any of the participating cores takes only one cycle.
- **Perfect-Mem**: In this mode, accessing a memory location on any of the participating cores takes one cycle plus the hit or miss time of the corresponding cache line.
- **Perfect-Operand**: Intra-block instruction and distributed protocol communication takes only one cycle.
- **Perfect-All**: This mode combines the *Perfect-Reg*, *Perfect-Mem*, and *Perfect-Operand* modes.

Figure 8 shows the performance of the flat, deep, and adaptive mapping strategies running with 16 dual-issue cores.

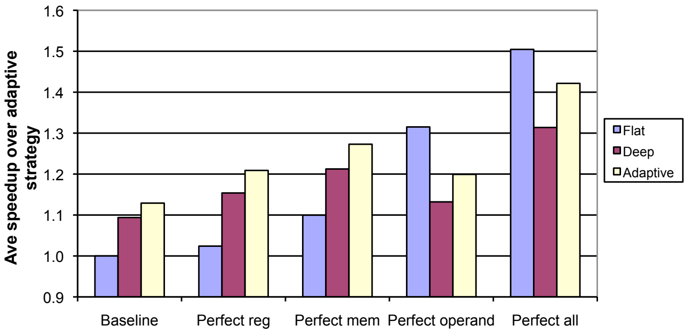


Figure 8. Average speedup over the adaptive strategy for SPEC benchmarks (fp and int) running on 16 dual-issue cores with perfect register, memory and intra-block instruction and distributed protocol communication.

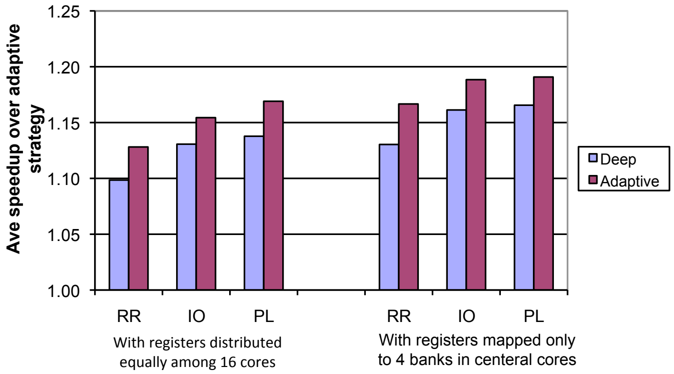


Figure 9. Speedup over the adaptive strategy for SPEC benchmarks running on 16 dual-cores with different block selection algoirithms used by the block mapper (round-robin, inside-out and preferred-location).

If all register accesses could be local, the performance of the flat mapping strategy changes slightly but the speedup of the deep and adaptive strategies improves by about 6%, as shown in the *Perfect-Reg* bars. Perfect memory accesses improve the speedup of all three strategies by about 11% for the deep and adaptive strategies. These results show that reducing the inter-block communication overhead caused by register and memory accesses can improve performance significantly for the deep and adaptive mapping strategies. Localizing all intra-block communication, as shown in the *Perfect-Operand* bars, changes the speedup of the deep and adaptive strategies only slightly. The speedup of the flat mapping strategy, however, improves by about 30% and outperforms the deep and adaptive strategy by at least 15%.

5.3. Reducing the Inter-block Communication Overhead in the Deep and Adaptive Strategies

We implemented the inside-out and preferred-location core selection algorithms discussed in Section 4.4. Figure 9 shows performance results for the round-robin (RR), inside-out (IO) and preferred-location (PL) core selection algorithms running on 16 dual-issue cores. These core selection algorithms offer modest improvements.

Another way to reduce communication overhead between blocks is to map registers to the cores close to the center of the execution substrate. This mapping could be achieved by modifying the register allocator to give priority to cores close to the center for the most critical registers. This solution would have the undesireable side effect, however, of requiring the register allocator to make additional assumptions about the underlying substrate. Instead, we use a hash function implemented in the TFlex simulator to map all architectural registers to the cores near the center. This approach may require larger register files in the central cores, which may not be practical. Also, programs with high register bandwidth requirements may suffer due to contention on the cores near the center. We chose to implement this test in the hardware, however, because doing so allowed the compiler to remain agnostic to the layout of the underlying substrate, and it did not require recompiling or modifying the compiler. Figure 9 shows performance results for different core selection algorithms with the registers mapped to the central cores.

Using the inside-out or preferred-location algorithms instead of round-robin improves the performance of the deep and adaptive strategies. This performance improvement is more significant when the microarchitecture maps the registers only to the four cores located in the center of the sixteen-core array. The preferred-locations algorithm, when used with the restricted register mapping, achieves the best speedup for both the deep and adaptive strategies (the two right-most bars in Figure 9). This speedup is about 3% higher than the round-robin algorithm without the restricted register mapping (the two left-most bars in Figure 9).

Figure 10 shows the performance achieved using the adaptive and deep mapping strategies before and after reducing the register and memory communication overhead for the individual SPEC benchmarks. These results are normalized to the flat mapping strategy. In this figure, *Deep* and *Adaptive* represent the deep and adaptive strategies using the round-robin core selection algorithm and *PL Deep* and *PL Adaptive* represent the deep and adaptive strategies using the the preferred-locations core selection algorithm with the registers mapped to four central cores. When using the round-robin algorithm, the deep and adaptive strategies outperform flat by 9% and 13% on average, respectively. These speedups increase by 5% when using the preferred-location core selection algorithm and the restricted register mapping.

5.4. Communication Overhead

We measured the communication overhead for each mapping strategy by counting the number of communication

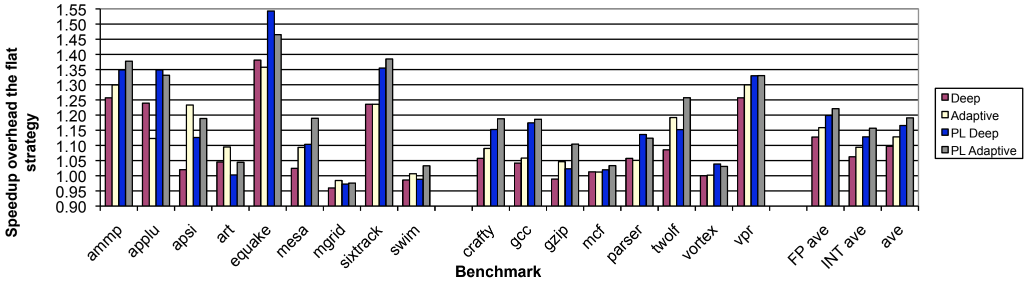


Figure 10. Speedup over the flat mapping strategy for the SPEC benchmarks with 16 dual-issue cores using the pref-location algorithm.

hops necessary for each register access, memory access, and operand bypass. Figure 11 shows the average communication overhead for each block mapping strategy for the SPEC benchmarks running on 16 dual and single-issue cores. The bars for single and dual-issue cores are labeled *DI* and *SI*, repectively. These results are normalized to the total hop count when using the flat mapping strategy on dual-issue cores. Using the flat mapping strategy, 70% of communication consists of operand transfer and distributed protocols among the cores. With the deep and adaptive mapping strategy these values become 9% and 12%, respectively, when runnig on dual-issue cores. Memory accesses cause almost the same amount of traffic for all three mapping strategies, but the overhead of register accesses is reduced for the flat mapping strategy. The static instruction scheduling algorithm considers the location of registers on an abstract substrate when calculating the placement cost for each instruction for flat mapping, thus minimizing register latency. When running on single-issue cores, the network traffic does not change for the flat and deep strategies. The adaptive block mapping strategy, however, shows an increase in the operand traffic, which is relatively small compared to the operand traffic of the flat strategy.

The PL selection algorithm with all registers located on the four central cores reduces register traffic to almost half of its previous value for the deep and adaptive strategies, the *PL deep* and *PL adaptive* bars in Figure 11. On average, the PL deep and adaptive stragies have 70% lower communication overhead than the flat strategy. This improvement should translate to a significant reduction in power consumption caused by on-chip network transactions.

5.5. The Effect of Instruction Criticality

This section investigates the extent to which the accuracy of the criticality bits affects performance. Because instructions within a core can issue out of order, the instructions' order in the reservation stations only matters when multiple instructions are ready to execute at the same time, and a more critical instruction gets lower priority. Thus, conservatively using more locality bits and fewer criticality bits may be the right choice.

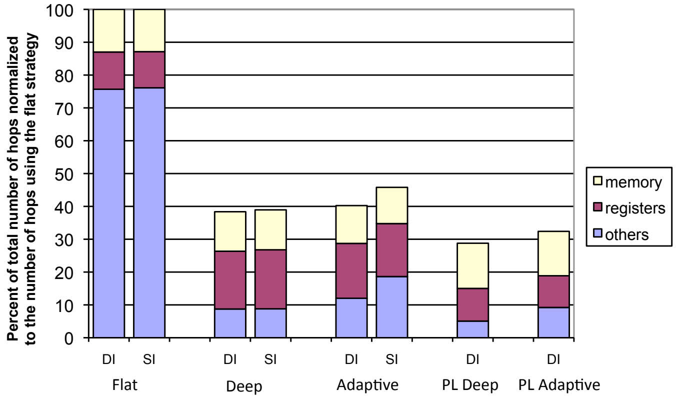


Figure 11. Communication overhead in terms of hop count for the SPEC benchmarks running on 16 single (SI) and dual-issue (DI) cores.

To investigate the effect of criticality bits, we test various bit distributions with the deep strategy. We selected the deep strategy for this experiment because this strategy uses instruction identifiers entirely for criticality information. First, we test binaries scheduled specifically for the deep strategy, in which all seven bits of the IDs are used for criticality information. We also run the deep strategy with binaries scheduled for an extreme flat mapping with blocks mapped to 32 cores. In these binaries, five bits of the instruction identifiers are used for locality information, and only two bits are used for criticality information.

Table 2 compares the performance using these two sets of binaries for the deep strategy on the SPEC benchmarks run-

Table 2. The performance of the deep block mapping strategy using different number of criticality bits.

	2-bit cr.	7-bit cr.	reversed cr.
1 dual-issue core	1.00	1.09	0.86
4 dual-issue cores	2.55	2.61	2.49
16 dual-issue cores	3.84	3.86	3.84

ning on different numbers of dual-issue cores. It also includes the performance achieved using the adaptive strategy on these benchmarks. The results in this table are normalized to the performance of the benchmarks running on one core with two criticality bits. Running on one core, the deep strategy using 7-bit information performs 9% better than the deep strategy with 2-bit criticality information. This improvement decreases to 2% and 0.5% percent when running on 4 and 16 cores, respectively. One reason why the criticality becomes less important may be that more parallelism is possible between blocks. To further investigate this observation, we modified the simulator such that instructions are dispatched in the reverse of their criticality order set by the compiler. As shown in the table, this change causes major performance loss only when running on one core.

5.6. Comparison with Conventional Processors

A performance comparison with production ISAs is beyond the scope of this paper. In addition, a relevant comparison requires factoring in the differences due to the ISA, memory system, cycle time, and different compilers. A related study [1] compares TRIPS with the Core2Duo in terms of cycle count achieved when running the same benchmarks used in this work. That study [1] indicated that TRIPS outperforms Core2Duo by 50% on the EEMBC benchmarks. For SPEC FP, TRIPS and the Core2Duo achieve similar cycle counts, but Core2Duo outperforms TRIPS by 40% when running SPEC INT. Using the flat block mapping strategy, TFlex outperforms TRIPS by 19% when running on eight TFlex cores. This speedup increases to 42% when using the best per-application TFlex configuration [1].

6. Discussion

The analysis of how to map instructions to clusters may be relevant for distributed processing in production ISAs, as well. If future processors perform clustering on a chunk-by-chunk basis they can benefit from these results, which show how to map the chunks to clusters. Because conventional ISAs do not have block headers, they will require a different mechanism to convey per chunk mapping information (e.g., hint instructions).

The idea of encoding locality and criticality information into the ISA can be applied to other composable multi-core systems. These systems should support varying numbers of cores and system configurations. Having locality and criticality information encoded in the binary facilitates hardware runtime decisions in such systems.

Measuring static concurrency of a code region at compile time, and using that concurrency to choose resources for that region at runtime, may help future systems with RISC or CISC ISAs as well. In an SMT processor, issue bandwidth can be assigned to each thread running in the system according to a compile-time-evaluated concurrency value associated with that thread. For these systems, however, different concurrency evaluation functions may be needed.

7. Conclusions

This paper explores various strategies to dynamically map blocks of instructions to a distributed hardware substrate consisting of composed cores acting as a single processor. A run-time block mapper, implemented in hardware, maps instructions to cores. We explore a spectrum of *fixed* policies, in which the block mapper maps each block of instructions to the same number of cores. At one extreme, a *flat* mapping policy partitions the instructions in each block among all participating cores, emphasizing intra-block parallelism, but increasing intra-block communication. At the other extreme, a *deep* mapping policy maps all of the instructions in a block to a single core, but successively maps blocks to different cores. The deep strategy minimizes intra-block communication delays, but allows no intra-block parallelism beyond the issue width of the individual cores, and makes inter-block communication more expensive.

For single-issue cores, a flat mapping policy is the highest-performing fixed choice. Although the flat mapping policy increases the processor's complexity and communication overheads, single-issue cores need the additional intra-block concurrency that the flat mapping provides. The low additional complexity of dual-issue cores, however, harvests enough of the intra-block parallelism to change the ideal mapping to a deep mapping. The deep mapping eliminates substantial intra-block operand communication, and the dual-issue cores provide enough intra-block parallelism that a flatter mapping provides no benefit. Both of these policies are limited, however, because they are fixed: each block is mapped to the same number of cores, regardless of the variance in ILP across different blocks.

Using block-level concurrency information provided by the compiler, the block mapper can specialize its policies on a per-block basis and harvest more performance than is possible using the fixed policies. For single-issue cores, this adaptive policy may be a good design choice because it exploits intra-block concurrency while limiting operand network traffic, which is a significant source of energy overhead for this class of architectures. As the issue width of the individual cores increases, however, the benefit of a statically guided adaptive policy decreases. Dual-issue cores can exploit enough parallelism locally, without any communication overhead, that the added complexity of the adaptive policy may not be worth the corresponding performance improvements. If the issue width of individual cores increases further, the adaptive policy's utility will continue to decrease. However, it is likely that more flexible and sophisticated mapping policies may increase performance and reduce energy further, even for dual-issue cores.

Acknowledgments

We would like to thank Paul Gratz, Bert Maher and Sadia Sharif for their helpful feedback. This work was supported by NSF grant EIA-0303609 and CSR-0615104, DARPA contract F33615-03-C-4106, and an NSF Graduate Fellowship.

References

[1] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, “Composable lightweight processors,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture*. Chicago, Illinois, USA: IEEE Computer Society, 2007, pp. 381–394.

[2] M. D. Hill and M. R. Marty, “Amdahl’s law in the multicore era,” in *IEEE Computer (to appear)*, 2008.

[3] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, “Core fusion: accommodating software diversity in chip multiprocessors,” in *34th annual international symposium on Computer architecture*, San Diego, California, USA, 2007, pp. 186–197.

[4] H. Zhong, S. A. Lieberman, and S. A. Mahlke, “Extending multicore architectures to exploit hybrid parallelism in single-thread applications,” in *IEEE 13th International Conference on High Performance Computer Architecture*, Phoenix, Arizona, 2007, pp. 25–36.

[5] D. Tarjan, M. Boyer, and K. Skadron, “Federation: Out-of-order execution using simple in-order cores,” University of Virginia, Department of Computer Science, Tech. Rep. CS-2007-11, August 2007.

[6] 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*, vol. 30, no. 9, 1997.

[7] V. Zyuban and P. Kogge, “Optimization of high-performance superscalar architectures for energy efficiency,” in *2000 international symposium on Low power electronics and design*, Rapallo, Italy, 2000, pp. 84–89.

[8] R. Canal, J. Parcerisa, and A. Gonzalez, “Dynamic cluster assignment mechanisms,” in *6th International Symposium on High Performance Computer Architecture*, Toulouse, France, 2000, pp. 133–142.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective superscalar processors,” *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 206–218, 1997.

[10] H.-S. Kim and J. E. Smith, “An instruction set and microarchitecture for instruction level distributed processing,” in *29th Annual International Symposium on Computer Architecture*, Alaska, 2002.

[11] K. L. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, “The multicluster architecture: Reducing processor cycle time through partitioning,” in *30th Intl. Symposium on Microarchitecture*, North Carolina, USA, 1997, pp. 327–356.

[12] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, “Multiscalar processors,” in *22nd annual international symposium on Computer architecture*, Barcelona, Spain, 1995, pp. 521–532.

[13] V. Krishnan and J. Torrellas, “A chip-multiprocessor architecture with speculative multithreading.” *IEEE Trans. Computers*, vol. 48, no. 9, 1999.

[14] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21 – 29, 2005.

[15] P. Salverda and C. Zilles, “Fundamental performance challenges in horizontal fusion of in-order cores,” in *International Symposium on High-Performance Computer Architecture*, 2008, pp. 252–263.

[16] 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 *30th Annual International Symposium on Computer Architecture*, San Diego, CA, june 2003, pp. 422–433.

[17] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin, “WaveScalar,” in *36th Symposium on Microarchitecture*, December 2003.

[18] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley, “Compiling for EDGE architectures,” in *International Symposium on Code Generation and Optimization*, Manhattan, NY, Mar. 2006.

[19] 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 *25th annual international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 45–54.

[20] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha, “A spatial path scheduling algorithm for EDGE architectures,” in *ASPLOS-XII*, 2006, pp. 129–140.

[21] C. Kim, D. Burger, and S. W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” in *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002, pp. 211–222.

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

[23] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *10th International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001.