Balancing Local and Global Parallelism for Single-Thread Applications in a Composable Multi-core System

Behnam Robatmili Katherine Coons
Doug Burger
University of Texas at Austin, Computer Science Department
{beroy, coonske, dburger}@cs.utexas.edu

Abstract

One way to improve the performance of a single-threaded application is to run regions of the code speculatively on a multi-core system. In a composable multi-core system, resources such as the L1 cache, register file, and instruction window are distributed among the cores. In such a system, there are many ways to map instructions to the cores. At one extreme, instructions in each code region are distributed evenly across all of the cores, favoring *local parallelism* within the region, but also increasing *local communication.* At the other extreme, instructions in each region are assigned to a single core. This strategy minimizes *local communication* within the code region, but at the cost of *local parallelism.* We propose a hybrid strategy that uses a compile time critical path analysis to determine the available concurrency in the region of code. A block mapper uses this concurrency value to decide the number of cores to which the region of code should be mapped. Our results show that this method improves performance significantly and reduces communication among cores.

1 Introduction

Balancing concurrency and communication is one of the fundamental challenges in parallel processing. Exploiting fine-grained concurrency brings this problem to the forefront because fine-grained concurrency is often coupled with fine-grained communication, and it is more difficult to amortize the overheads of communication at a fine granularity. To balance concurrency and communication at a fine granularity a system can use both dynamic information, gathered by the hardware, and static information, gathered by the compiler. Which method is preferable, hardware or software, varies with the hardware and the execution model. Different distributed architectures use different breakdowns between hardware and software approaches.

RAW [20] favors an approach that relies heavily on the compiler. The RAW compiler schedules instructions in time to exploit concurrency, and places instructions on a physical substrate. The TRIPS [16] compiler forms predicated regions called hyperblocks and places instructions in each hyperblock on a grid of 16 ALUs, where they are issued dynamically. Wavescalar [19] is a dataflow processor that uses static placement of instructions on an hierarchical substrate. The compiler for Multicluster [7] processors partitions instructions between clusters during register allocation to minimize remote register accesses. Instructions in each cluster are scheduled dynamically by the hardware. In Thread-Level Speculation [14], the hardware automatically spawns speculative threads, selected by the compiler, on multiple cores. Multicluster superscalar processors [21] rely on the hardware to dynamically steer instructions to different clusters based on the dependencies between instructions. 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 [15]. The ISA for Instruction Level Distributed Processing [13, 12] 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.

We propose a model for mapping blocks of instructions to a distributed substrate in which the compiler encodes the local (intra-block) concurrency and an abstract model of local communication in the ISA. Then, a block mapper chooses how to map that block to the distributed substrate in a way that minimizes global

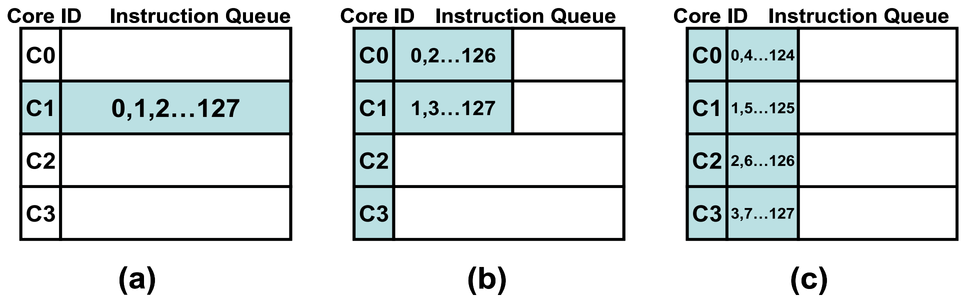


Figure 2: Mapping an instruction block to various number of cores

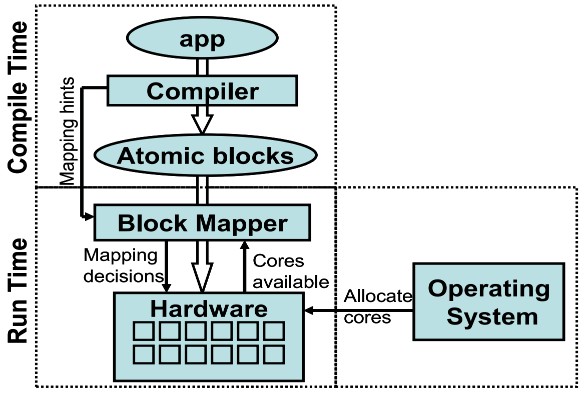


Figure 1: System Components

(inter-block) communication, exploits global concurrency, and attempts to accommodate the compiler's request for local concurrency. There is a large body of research on partitioning algorithms for multicluster processors [4, 7, 6, 3]. There are some similarities between those works and the work we present in this paper. This work, however, addresses partitioning and composablity at the same time.

We evaluate different block mapping strategies using TFlex [10], a composable lightweight processor that executes hyperblocks of instructions atomically on a distributed substrate. Figure 1 shows the components in a TFlex system. The TRIPS compiler [18] breaks the program into fixed-sized blocks of instructions. The EDGE ISA imposes several restrictions on 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. This 7-bit instruction identifier, implicitly encoded by that instruction's position in the block, determines on which participating core the instruction will be placed (depending also on the run-time mapping policy) and in what order that instruction will appear in the issue queue. The microarchitecture gives lower numbers higher issue priority within a single issue window. Figure 2 illustrates how the block mapper maps a block of instructions to different numbers of cores in an application running on four cores. In 2(a), the block mapper selects one core, core 1 in this example, on which to map the block. After fetching the block, the selected core prioritizes instructions in its issue queue based on their instruction identifiers. In 2(b), instructions are split between two cores selected by the block mapper, cores 0 and 1 in this example. Finally, in 2(c), the block mapper maps the instructions to all participating cores - each core executing 32 of total instructions in the block.

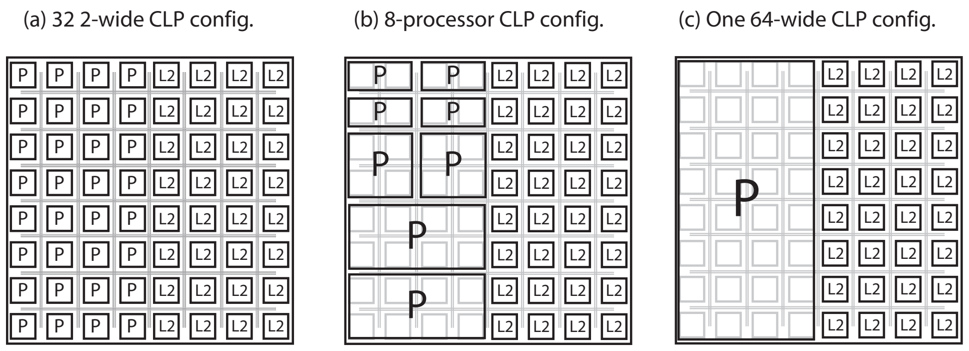
We discuss trade-offs between communication and concurrency, as well as the role of the compiler in each block mapping strategy.

2 Composable Lightweight Processors

To support workloads with differing degrees of parallelism, multi-core systems must adapt the number of cores [8]. 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 [9, 10].

TFlex is a composable lightweight processor (CLP) in which all microarchitectural structures, including the register file, instruction window, predictors, and L1 caches, are distributed across a set of cores [10]. 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, which are then treated by the program as a single processor. Figure 3 shows three of many possible configurations: 3(a) has 32 1-core processors, 3(c) has one 32-core processor, and 3(b) has a mix of processors composed of different numbers of cores.

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


(a) 32 2-wide CLP config.
(b) 8-processor CLP config.
(c) One 64-wide CLP config.
P
L2
Figure 3: Three CLP configurations [10].

holding that register in its register file [10].

For an application with N participating cores, the hardware fetches and executes up to N blocks in parallel. Inter-block communication occurs via registers and memory, which are distributed across the cores based on reconfigurable hash functions. In this paper, we study the block mapping strategy, focusing on how the block mapper can make use of information provided by the compiler. The abstract concurrency and the core selection strategies proposed in this paper can easily be applied to other distributed processors. An EDGE ISA offers an advantage, however, because the criticality and locality information can be implicitly encoded via each instruction's location within the block, without any overhead in terms of code size.

3 Block Mapping Strategies

In this section, we discuss three different block mapping strategies for the TFlex composable processor. A block mapping strategy must balance communication and concurrency at both the global and the local level. For example, assigning a block to fewer cores may be beneficial to limit local communication, but not if doing so reduces the local concurrency enough to outweigh the benefit. If sufficient global concurrency is available, however, then reducing the number of cores may decrease local communication without any cost in total concurrency, as the global concurrency can compensate for the loss of local concurrency. If the associated cost in global communication is too high, however, then the benefit may decrease. We evaluate two strategies in which the number of cores that a block is mapped to is fixed. These strategies provide all blocks with the same opportunities and limitations in terms of concurrency and communication. Then, we propose a third mapping strategy in which we allow the block mapper to vary the number of cores used to execute each block.

3.1 Flat Mapping

With a flat mapping strategy, the block mapper distributes instructions for every block across all participating cores using instruction identifiers set by the compiler. For example, when running on four cores, the flat mapping strategy maps all blocks as shown in Figure 2(c). This approach attempts to exploit as much local concurrency as possible, but at the cost of increased local communication. This strategy relies heavily on the compiler, and the block mapper's job is very simple.

Static Assignment of Instruction IDs: Once the compiler forms hyperblocks, the instruction scheduler's job is to place the instructions within a hyperblock onto the cores in a way that minimizes communication delays, yet exploits the available concurrency. We use the spatial path scheduling algorithm designed for EDGE architectures [5] to assign instruction IDs to instructions. For each instruction, the best core is the location at which the placement cost is the lowest. Among all of the instructions under consideration, the one with the largest minimum placement cost is the most critical, and should be placed next. The placement cost is calculated based on various features of the instruction, the core under consideration, the data flow graph containing the instruction, and the location of the registers used in the hyperblock.

3.2 Deep Mapping

With a deep mapping strategy, the block mapper maps all instructions within a block to a single core. Figure 2(a) shows an example of the deep mapping strategy on a block when running on four cores. This strategy relies more heavily on the block mapper than on the compiler. The block mapper can balance global concurrency and communication by selecting intelligently to which core the next block should be mapped.

The deep mapping strategy may increase global communication, because cache banks and registers are distributed across the cores, while each block executes on only a single core. By mapping all instructions within

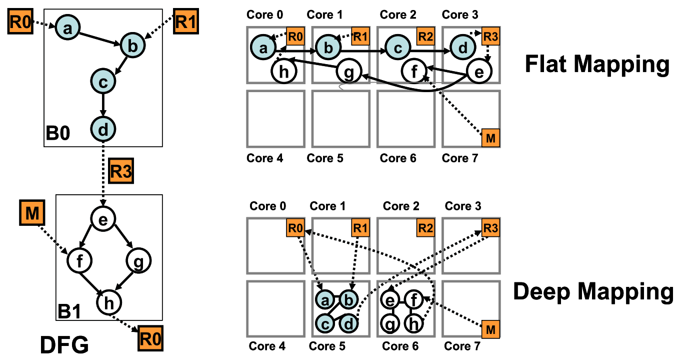


Figure 4: A sample DFG consisting of two hyperblocks mapped using the flat and deep mapping strategies. Solid and dotted lines represent local and global communication, respectively.

a block to a single core, the deep mapping strategy limits the local concurrency to the issue width of the core. Because multiple blocks are mapped onto the grid at once, however, if sufficient global concurrency is available, then the total concurrency in the system will not decrease.

Figure 4 provides a simple example of the flat and deep mapping strategies for two blocks, $B0$ and $B1$, on a 16-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 $B0$ also loads from a cache bank M located on core 7. The value communicated between blocks $B0$ and $B1$ via register $R3$ is an example of global communication, while the value produced by instruction a and consumed by instruction b is an example of local communication. With flat mapping, the instruction scheduler is able to place all instructions that access registers on the same core as the corresponding register. With deep mapping, however, the blocks are mapped to cores dynamically in a round-robin fashion, so most register accesses go to remote cores.

3.3 Adaptive Mapping

The flat and deep mapping strategies both suffer from the fact that the block mapper maps all blocks within a given application to the same number of cores, C. With the flat mapping strategy, $C = N$, where N is the number of cores participating in executing the application. With deep mapping, $C = 1$. As a result, the flat mapping strategy may under-utilize cores or cause excessive communication overheads if the block contains little concurrency, and the deep mapping strategy may fail to take advantage of local concurrency if the block contains large amounts of concurrency. A mapping strategy that can select a different number of cores for each hyperblock based on its communication patterns and available concurrency may be desirable.

We propose a block mapping strategy that adaptively selects the number of cores for each block. The compiler encodes the available concurrency in the block header, and supplies information about local communication by assigning instruction IDs to the instructions in the block. At runtime, the block mapper dynamically selects a set of cores for the block based on the local concurrency value provided by the compiler. For example, when running on four cores, the block mapper may choose to map each block to one, two or four cores (see Figure 2).

The algorithm for choosing which cores to map the next block to is round-robin, but modified to account for blocks that request varying numbers of cores. If there is not enough room in the instruction window for the next block, then instruction fetch will be stalled until there is sufficient space available. More sophisticated algorithms for dynamic mapping are possible, but might make the hardware implementation impractical. Round-robin strategies may be desirable because they can be implemented in a distributed fashion without any centralized components.

Calculating local concurrency: The instruction scheduler computes the critical path length through the block and uses it to compute the local *concurrency* in the block:

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

where, $BlockInstructionCount$ and $CriticalPathLength$ are the total number of instructions in the block and the length of the critical path through the block in cycles, respectively. This metric gives an estimate of the IPC available inside the block. For example, if a block has a long and narrow chain of dependent instructions, then the concurrency value may be smaller than one.

At runtime, the block mapper chooses the cores needed for each block as follows:

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

where, $IssueWidth$ is the issue width of each core in the system. The block mapper uses this number of cores to map the block if enough resources are available.

4 Results

We added support for these three mapping strategies to the validated TFlex simulator [10]. The microarchitectural parameters of each single core TFlex in our simulation are shown in Figure 1. We also modified the instruction scheduler for TFlex [5] to incorporate local concurrency information into the block header." We test each mapping strategy on the EEMBC [1] and SPEC [2] benchmarks. We use 29 EEMBC benchmarks with an iteration count of one hundred thousand. In addition, we use 9 intereger and 10 floating point SPEC

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 [11] (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

Table 1: Single Core TFlex Microarchitecture Parameters [10]

benchmarks with the reference (large) dataset simulated with single simpoints [17].

4.1 Performance

Figures 5 and 6 show performance using the flat, deep, and adaptive mapping strategies for the EEMBC and SPEC benchmarks normalized to the performance of that benchmark on a single dual-issue core. We vary the number of cores allocated to the application from 1 to 32 cores. We also vary the issue width of the cores, using issue widths of one and two.

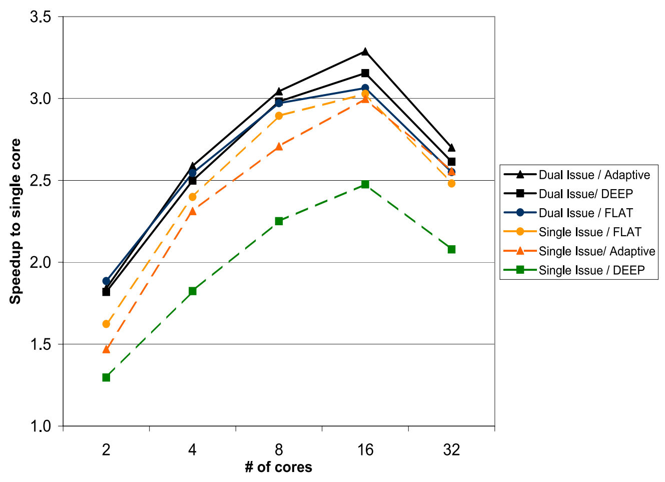
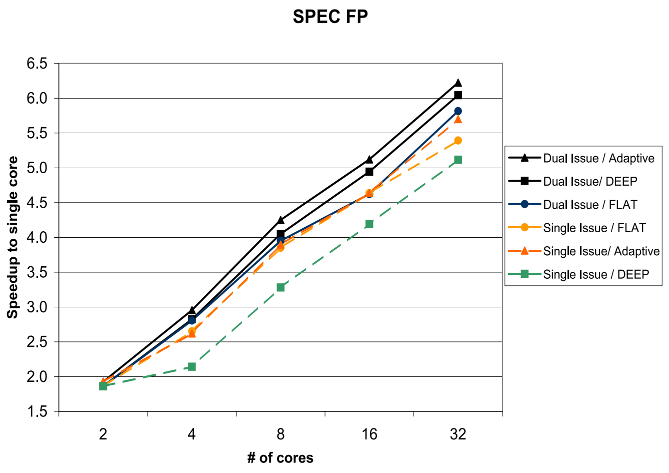


Figure 5: Average speedups over single core for EEMBC benchmarks varying the number of cores and the issue widths of the cores.

The EEMBC benchmarks reach their maximum performance when running on 16 cores using different mapping strategies but observe a significant slowdown with 32 cores. High operand network latency and insufficient branch prediction accuracy are responsible for this slowdown. With 32 cores, 31 blocks execute speculatively, and with low branch prediction accuracy, many of these blocks will be flushed. Programs with higher branch prediction accuracy will be better able to take advantage of 32 cores, because they are better able to use global concurrency to keep the 32 cores busy when sufficient local concurrency is not available. The SPEC integer benchmarks also perform worse with 32 cores than 16 cores. The performance of the SPEC floating point benchmarks improves when running with 32 cores, however, which is probably due to high branch prediction accuracy for the SPEC floating point benchmarks.



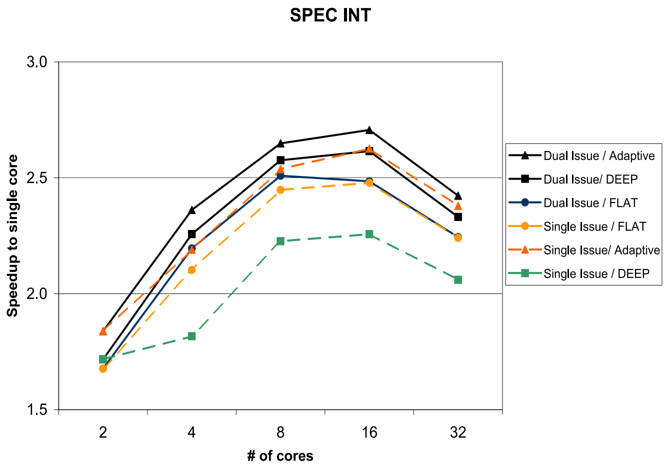


Figure 6: Average speedups over single core for SPEC benchmarks varying the number of cores and the issue widths of the cores.

With dual-issue cores, the adaptive mapping strategy outperforms the other two strategies in all cases except the two-core case, in which the flat mapping strategy performs slightly better. With single-issue cores, the performance of the flat mapping strategy changes very little while both the deep and adaptive mapping strategies perform significantly worse. This drop in performance is more pronounced when using the deep mapping strategy because with a single-issue core, the deep mapping strategy is unable to exploit any local parallelism. While the deep mapping strategy loses performance when we move to single-issue cores, the adaptive mapping strategy is able to compensate for the loss in local parallelism by using more cores, as shown in Table 2, which indicates the percentage of executed blocks that use each number of cores with the adaptive mapping strategy for EEMBC benchmarks. With dual-issue cores, the adaptive mapping strategy maps more than 60% of blocks to only one core. When using single-issue cores, half as many blocks are mapped to a single core, and more than half of the blocks use two or four cores.

	1-core	2-core	4-core	larger
Single-Issue	63.6%	28.4%	6.5%	1.4%
Dual-Issue	29.9%	34.8%	26.9%	8.1%

Table 2: Average percentage of executed blocks with a specific recommended number of cores for EEMBC benchmarks running with single and double dual cores.

Figures 7 and 8 show the speedup achieved using the adaptive and deep mapping strategies normalized to the flat mapping strategy for individual EEMBC and SPEC benchmarks on 16 dual-issue cores.

For most programs the deep and adaptive mapping strategies outperform the flat mapping strategy. For some EEMBC programs, however, the flat mapping strategy is better. In *fft00*, for example, the inner-most block of the kernel reads eight registers from different banks and has four parallel high-latency multiply instructions. The deep mapping strategy suffers from high global communication and is also unable to extract the local parallelism within that block. With adaptive mapping, the block mapper chooses to map that block to two cores. This mapping allows the processor to extract some of the local concurrency available.

Figure 9 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 the top graph in the figure, the adaptive mapping strategy maps about 40% of blocks to two or four cores. For the SPEC integer benchmarks, the mapper maps 20% of the blocks to more than one core when using dual-issue cores, which indicates less concurrency within the blocks compared to the SPEC floating point benchmarks. When using single-issue cores, as shown in the bottom graph in Figure 9, half as many blocks are mapped to more than one core, and more than half of the blocks use two or four cores.

Some SPEC benchmarks including *ammp*, *equake*, *sixtrack*, and *vpr*, show significant speedups when using the deep and adaptive mapping strategies. We suspect this is because the most critical blocks in these applications have very little local 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 indicates that the instruction scheduler is able to find very little local concurrency in the most critical blocks of this benchmark, so a flat mapping strategy incurs extra local communication overheads without any benefit in local concurrency.

For most SPEC benchmarks, the adaptive mapping strategy performs better than the deep mapping strategy. The *bzip2* benchmark achieves the largest speedup using adaptive mapping, about 15%. For this benchmark, the block mapper chooses to map about 50% of blocks to four cores. This mapping suggests that there is a high amount of local concurrency available in this benchmark, and the adaptive mapping strategy is able to exploit this concurrency.

Figure 10 includes 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. The adaptive strategy outperforms both the deep and and the flat strategy in most cases.

These results suggest that the adaptive mapping strategy reduces the local communication significantly, and can extract both global and local parallelism. This strategy, however, suffers from global communication penalties due to dynamic block placement. We are looking at ways to reduce the global communication in this strategy for future work.

4.2 Communication Overhead

We measured the communication overhead for each mapping strategy by counting the number of communication 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-issue cores. These results are normalized to the total hop count when using the flat mapping strategy. Using flat mapping, 70% of communication is the result of operand transfer and distributed protocols among the cores. With the deep and adaptive mapping this value becomes 9% and 12%, respectively. Memory accesses cause almost the same amount of traffic for all three mapping strategies, but the overhead of register accesses is reduced for flat mapping. The static instruction scheduling algorithm considers the location of registers on the grid when calculating the placement cost for each instruction for flat mapping, thus minimizing register latency.

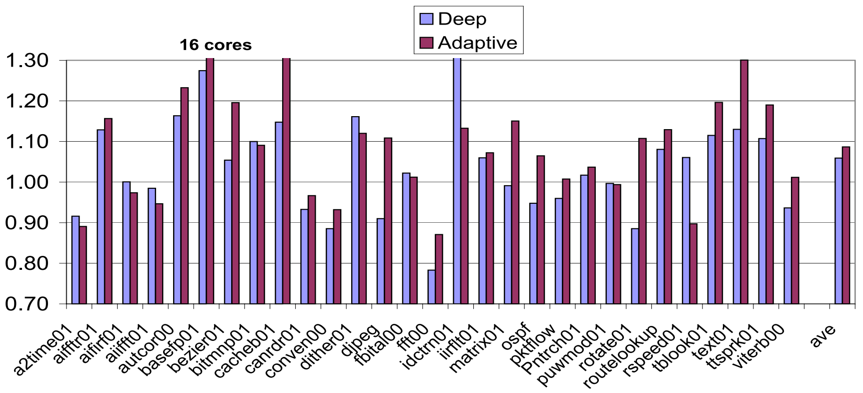


Figure 7: Speedup over the flat mapping strategy for the EEMBC benchmarks with 16 dual-issue cores.

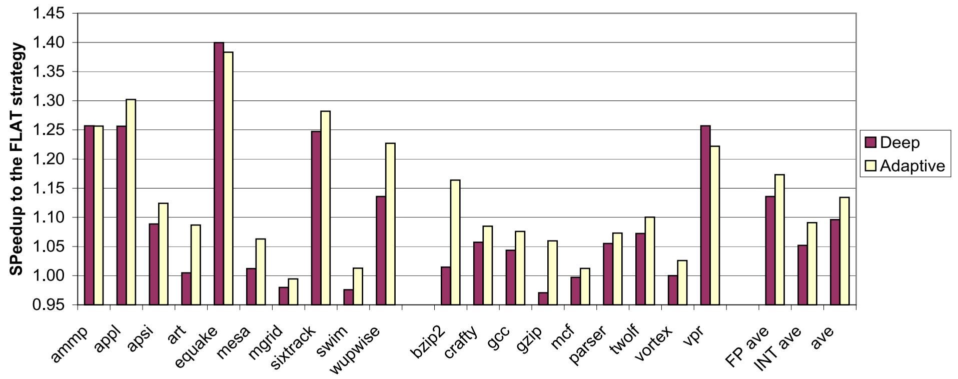


Figure 8: Speedup over the flat mapping strategy for the SPEC benchmarks with 16 dual-issue cores.

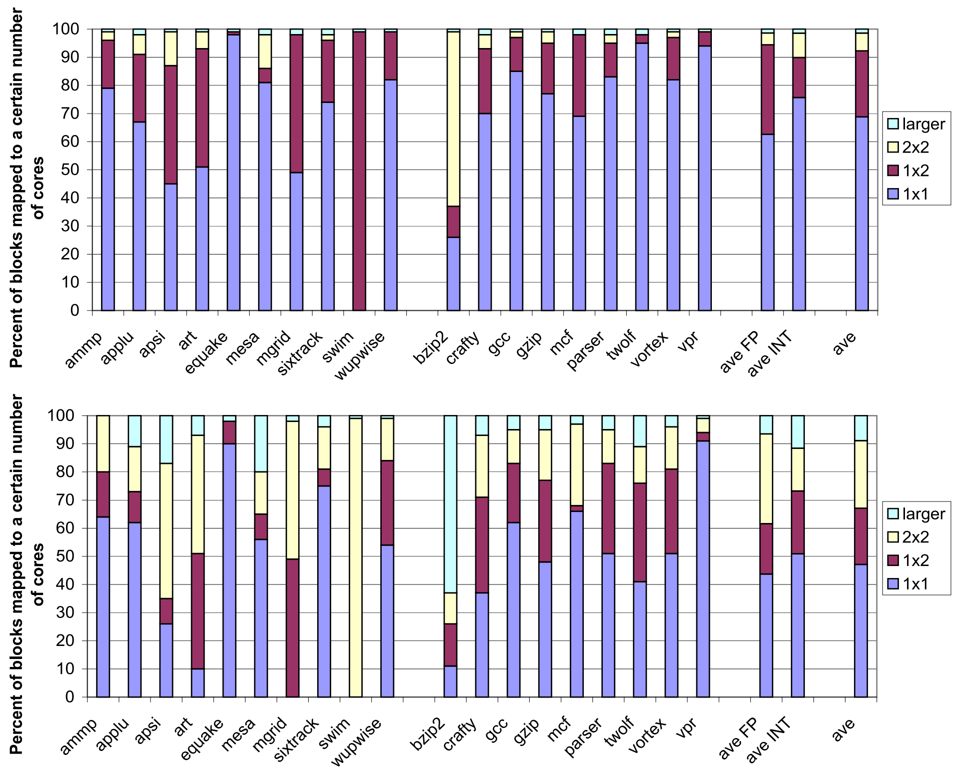


Figure 9: Percent of blocks mapped to a certain number of cores by the block mapper for SPEC benchmarks running with 16 dual-issue (top) and single-issue (bottom) cores.

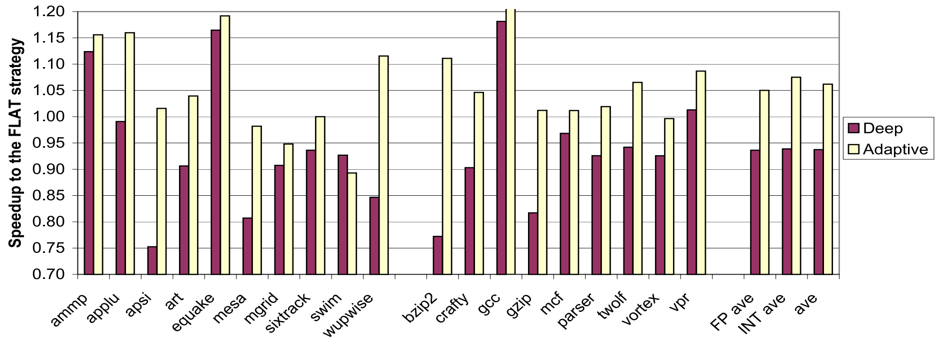


Figure 10: Speedup over the flat mapping strategy for the SPEC benchmarks with 16 single-issue cores.

We are looking at ways to reduce the global communication in this strategy for future work.

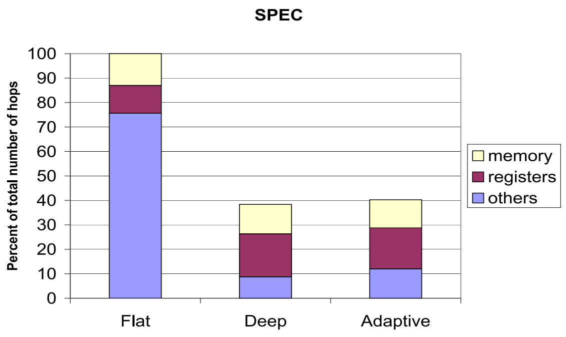


Figure 11: Communication overhead in terms of hop count for the SPEC benchmarks running on 16 dual-issue cores.

5 Conclusions

Future distributed processors must use a combination of hardware and software techniques to achieve high concurrency for different workloads while reducing the communication overhead. The adaptive block mapping strategy, proposed in this paper, uses abstract concurrency information provided by the compiler to balance the locality and criticality of instructions in each block. In addition to this information, the block mapper considers the hardware characteristics to choose appropriate number of cores to map each block. While achieving high performance, this method causes relative low communication over across th on-chip network.

References

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

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

[3] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *33rd International Symposiumon Microarchitecture*, page 337347, 2000.

[4] R. Canal, J. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. In *The 6th International Symposium on High Performance Computer Architecture*, page 133142, 2000.

[5] Katherine E. Coons, Xia Chen, Doug Burger, Kathryn S. McKinley, and Sundeep K. Kushwaha. A spatial path scheduling algorithm for EDGE architectures. In *ASPLOS-XII*, pages 129–140, 2006.

[6] E.Oezer, S. Banerjia, and T. Conte. A new approach to scheduling for clustered registerfile microarchitectures. In *The 31st International Symposiumon Microarchitecture*, page 308315, 1998.

[7] K. L. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing processor cycle time through partitioning. In *Proceedings of the 30th Intl. Symposium on Microarchitecture*, pages 327–356, 1997.

[8] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. In *IEEE Computer (to appear)*, 2008.

[9] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 186–197, New York, NY, USA, 2007. ACM.

[10] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Chankyu Kim, Doug Burger, and Stephen 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.

[12] Ho-Seop Kim and James E. Smith. Instruction level distributed processing. In *IEEE Computer 34, 4 (April)*, pages 59–65, 2001.

[13] Ho-Seop Kim and James E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *29th Annual International Symposium on Computer Architecture (ISCA'02)*, 2002.

[14] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Computers*, 48(9), 1999.

[15] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25(2):206–218, 1997.

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

[17] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, 2001.

[18] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *CGO06*, Manhattan, NY, March 2006.

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

[20] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9), 1997.

[21] V. Zyuban and P. Kogge. Optimization of high-performance superscalar architectures for energy efficiency. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, 2000.