The Good Block: Hardware/Software Design for Composable, Block-Atomic Processors

Bertrand A. Maher
Intel Corporation
bertrand.a.maher@intel.com

Katherine E. Coons Kathryn McKinley
University of Texas at Austin
{coonske,mckinley}@cs.utexas.edu

Doug Burger
Microsoft Research
dburger@microsoft.com

Abstract

Power consumption, complexity, and on-chip latency are forcing computer systems to exploit more parallelism efficiently. Explicit Dataflow Graph Execution (EDGE) architectures seek to expose parallelism by dividing programs into blocks of efficient dataflow operations, exposing inter and intra-block concurrency. This paper studies the balance of complexity and capability between EDGE architectures and compilers. We address three main questions. (1) What are the appropriate block granularities for achieving high performance efficiently? (2) What are good block instruction selection policies? (3) What architecture and compiler support do these designs require?

Our results show that the compiler requires multiple block sizes to adapt applications to block-atomic hardware and achieve high performance. Although the architecture for a single size is simpler, the additions for variable sizes are modest and ease hardware configuration. We propose hand-crafted and learned compiler policies for block formation. We find the best policies provide significant advantages of up to a factor of 3 in some configurations. Policies vary based on (1) the amount of parallelism inherent in the application, e.g., for integer and numerical applications, and (2) the available parallel resources. The resulting configurable architecture and compiler efficiently expose and exploit software and hardware parallelism.

1. Introduction

Limits on chip power consumption require that future improvements in computer system performance will come from efficient exploitation of parallelism, using a combination of hardware and software techniques. By providing support for coarse-grained atomic regions a processor can take advantage of parallelism at the level of regions rather than instructions. To gain this advantage, Explicit Dataflow Graph Execution (EDGE) architectures rely on the compiler to statically divide programs into blocks of dataflow instructions, which execute on parallel hardware [1, 4].

The compiler's ability to form large, effective blocks of instructions is critical to the performance of an EDGE architecture. Using 128-instruction blocks as in the TRIPS design, we find that even aggressive compiler algorithms fail to consistently fill large blocks because of fundamental structural constraints of programs [12]. While the compiler is sometimes able to fill large blocks, the total ratio of instructions to capacity at runtime remains relatively low and thus the system wastes time and resources on partially full blocks.

Given the difficulty of filling large blocks, we explore the design space of architectures with smaller block sizes, including the possibility of mapping multiple smaller blocks in place of large blocks. These experiments reveal two trends: (1) large blocks significantly improve the performance of compute intensive applications. These improvements come from parallelism within and between blocks due to the large window of execution presented in an EDGE design, and (2) smaller blocks generally outperform larger blocks on control intensive benchmarks. To take advantage of this dichotomy, we propose microarchitectural support for variable-size blocks, which enables "best-of-both-worlds" performance.

Because the compiler's optimization strategy changes with the introduction of variable-size blocks, we apply machine learning techniques to to construct optimized heuristics. We find that the best compiler policies are a function of core count. For one or two cores, the best performing policy does not use predication, thus conserving scarce resources. With more cores, the best policies aggressively predicate. These compiler policies together with variable-size block support support result in a performance improvement of up to a factor of 3 on single core configurations for microbenchmarks, over a factor of 2 on multiple cores compared to fixed-size blocks, and general purpose heuristics improve SPECINT programs by 35%.

2. Related Work

Related work exploits a combination of hardware and software techniques to increase the granularity of atomicity. Block-structured ISAs improve instruction fetch and issue bandwidth by aggregating instructions into atomic regions [5, 17, 16]. The compiler constructs enlarged blocks that contain a single path of control [15]. If an early exit

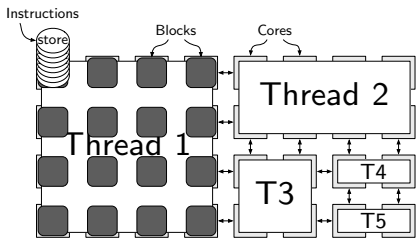


Figure 1. Mapping blocks to composable TFlex cores.

is taken from a block, the processor aborts and begins executing at the target of the early exit. While the goals of block-structured ISAs are similar to EDGE architectures, predication and dataflow in EDGE are significant architectural differences. Side exits from EDGE blocks do not abort execution, but predicate-out work. A side exit in a block-structured ISA causes the processor to squash the entire block and fetch a new block. EDGE relies less on dynamic branch prediction, but requires a wider machine to tolerate the overhead of predication.

VLIW compilers group independent instructions into atomic words [3]. Trace scheduling and superblock scheduling find instructions across basic block boundaries, which is useful when filling VLIW words [2, 7]. Another compiler approach for VLIW architectures is to form *hyperblocks*, single-entry, multiple-exit regions of predicated instructions [14]. Hyperblocks provide a scheduling region and framework for compiler heuristics to decide on the utility of predication. Both VLIW and EDGE are constrained by the number and type of instructions in a block [13, 22]. Because VLIW words however issue statically and in-order, the compiler must balance dependence chains. Instructions in EDGE blocks issue in dataflow order, which eases the compilation problem and makes large block sizes an effective design option.

3. EDGE Background

An EDGE processor fetches, issues, and commits blocks atomically. At least one block is non-speculative. Next-block prediction speculates subsequent blocks and multiple blocks are typically in flight. Blocks either commit entirely, or are flushed from the pipeline on misprediction. Each block contains a header that summarizes global communication and a body of instructions. The header indicates which global registers the block reads and writes. The processor uses the register information at the block level to rename registers when multiple blocks are in flight.

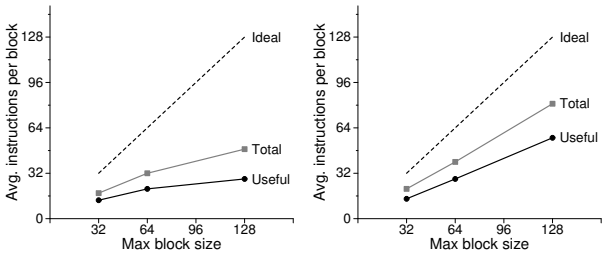
Within a block, instructions communicate directly using a target dataflow format. Dataflow communication is key to scalability. Because the ISA uses speculation to execute multiple blocks at once and a hierarchical namespace—global registers between blocks and temporary dataflow names within blocks—the processor efficiently supports a large number of instructions in flight. Register reads and writes are the global communication mechanism between blocks. RISC and CISC ISAs globally communicate with registers, but at the granularity of a single instruction and each access is through a shared register file and rename table. Point-to-point communication within a block eliminates all shared structure accesses and provides scalability to block-atomic architectures.

Prior EDGE designs have used fixed-size blocks simplify instruction cache and issue queue design. Each block is fully cache-resident or not and occupies a fixed set of lines indexed by a single tag. The header and instructions are at fixed known locations. Since a single cache tag corresponds to an entire block, hit detection is fast. On a miss, the cache evicts the entire victim block and fetches the new block from a lower level of the memory hierarchy. In the issue queue, each core has sufficient slots to accommodate all the instructions in a block. Instructions within the queue are located simply by offset. Register renaming is performed at the block granularity, indexed by block, which reduces the amount of renaming hardware to a function of the maximum number of blocks in flight.

3.1. EDGE Support for Composability

Dynamic multicore processors, which adapt their parallel resources to the workload at hand, provide the best performance tradeoff given a mix of sequential and parallel work [6]. EDGE allows dynamic composition of cores at a block granularity—one or multiple cores can execute a single block [10, 19]. Software configures the number of cores to match the workload: more cores per thread can accelerate blocks of sequential code, while fewer cores per thread with multiple threads in flight can accelerate parallel workloads. While composability can be achieved using a RISC or CISC ISA, as in Core Fusion [8], fine-grain register communication and frequent control decisions limit composability to a small number of cores. Coarse-grain block communication makes EDGE architectures more scalable.

This paper evaluates performance using the TFlex microarchitecture simulator, which models a composable EDGE chip multiprocessor consisting of moderately powerful cores [10]. Figure 3 shows how TFlex composes multiple cores to accelerate the execution of a single thread, e.g., Thread 1, and executes multiple threads in parallel. Within a thread, TFlex dynamically maps blocks to one or more cores. Each core's instruction queue has a fixed size, and can accommodate an entire block or a fraction of a block. Robatmili shows mapping one block to each core often performs best and we use this configuration as our baseline [19]. For example, Thread 1 has 16 participating cores. The default maps one block to each core, and thus up to 16 blocks are in flight, which exploits medium-grain concurrency between blocks. For a fixed-size block, the num-



(a) SPECINT (b) SPECFP

Figure 2. Dynamic average total and useful instructions per block with various maximum block sizes.

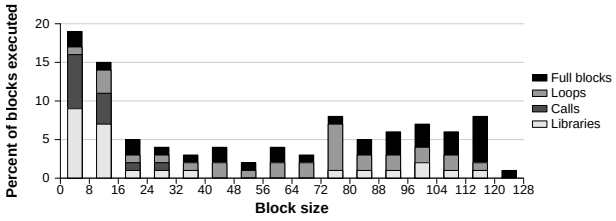


Figure 3. Distribution of block sizes in SPEC CPU2000 integer benchmarks, weighted by execution frequency. The categories indicate why the compiler could not merge that block with the next one in the execution trace.

ber of participating cores defines the number of instructions per block on each core.

3.2. EDGE Compilation

The compiler's block formation strategy exposes concurrency and amortizes overheads by forming large blocks. Prior work shows that iterative block formation solves phase ordering problems between if-conversion, loop transformations, and scalar optimizations to produce high-performing blocks [13]. Iterative block formation includes scalar optimizations, such as global value numbering [21], predicate minimization [23], and peephole optimizations.

During block formation the compiler repeatedly decides which (if any) of the successors blocks to merge. The compiler uses a heuristic function that selects the most desirable next block. When merging blocks the compiler performs if-conversion as necessary, adding predication, and code duplication, ensuring the block has a single entry point. The compiler performs scalar optimizations after each merge and then ensures it meets the architectural constraints on block size, register reads and writes, loads, and stores. If the block exceeds the constraints, the compiler discards the merge and chooses another next block, if one exists.

4. Block Size Analysis

To generate high-performance code for an EDGE processor, a compiler must construct large blocks of instructions. Larger blocks allow the processor to form a larger instruction window, which increases available instruction-level parallelism and improves latency tolerance. Furthermore, if the processor caches fixed-size blocks, small blocks can worsen L1 instruction cache pressure. The TRIPS processor, which caches fixed-size, 128-instruction blocks, suffers upwards of ten instruction cache misses per 1,000 instructions on several SPECINT benchmarks [4, 10].

Setting a smaller architectural block size reduces, but does not eliminate this underutilization problem. As Figure 2 shows, the dynamic average number of instructions per block does increase almost linearly with increasing architectural block size. On SPECINT, average block fullness declines slightly as the architectural size increases, and on SPECFP, fullness remains unchanged regardless of size. The fraction of *useful* instructions—those which contribute to the block output—decreases somewhat with larger block sizes, owing to the larger number of speculative instructions the compiler includes to fill larger blocks.

To explain the observed block sizes, we examine the distribution of block sizes at runtime. Figure 3 shows a dynamically weighted histogram of block sizes for the SPEC CPU2000 benchmarks compiled with a maximum block size of 128 instructions. Small blocks less than 16 instructions constitute over 35% of total blocks; blocks between 16 and 72 instructions constitute 25% of the total; and large blocks over 72 instructions are the remaining 40%. This skewed distribution is explained by the compiler's conditions for block termination:

Calls. Function calls end EDGE blocks because blocks cannot be re-entered upon return. While aggressive inlining reduces the frequency of calls, they remain a significant obstacle. The results in Figure 3 allow up to 100% code growth from inlining, which is large for typical compilers.

Libraries. We treat calls into libraries separately, since they are not candidates for inlining in our compiler. Library calls constitute a significant fraction of the smallest blocks, with over 8% of dynamic blocks being library calls with fewer than eight instructions.

Full Blocks. The compiler cannot merge blocks if doing so would violate an architectural constraint, such as block size or number of memory operations.

Loops. Similarly, unrolling or peeling a loop cannot exceed the block constraints. While loops could be unrolled past a single block to better align with the block size, we do not include this optimization because it achieves limited speedup in practice [18] and increases the implementation complexity of block formation.

The high proportion of small blocks that cannot be merged due to fundamental structural constraints in the code indicates that the compiler cannot solve this problem alone. The most viable approach is architecture and compiler support for variable-size blocks because the compiler *can*, and in many cases does, form large block. However,

the benefits of these large blocks are offset by scenarios in which the compiler is fundamentally limited due to structural constraints. Variable-size blocks allow the compiler to exploit the advantages of large blocks when possible, but without wasting resources when it cannot fill a large block.

5. Variable-Size Block Architecture

The TFlex microarchitecture as originally designed maps one block per core, which requires each core to have control and renaming logic for only one block at a time. Supporting multiple blocks on a single core requires additional support from the instruction queue, renaming logic, block control logic, and instruction cache. The renaming and block control logic must be larger and more associative, but are similar in design. The instruction cache and issue queue, however, must be modified as described below.

Instruction Cache. The original TFlex microarchitecture uses an instruction cache design that was highly specialized for the large, fixed-size blocks. The design employed separate structures for block headers and a block's instructions, and tuned block size to hold entire blocks, replacing blocks at a single-block granularity [10]. With variable-sized blocks, handling variable sizes and small blocks efficiently works poorly in the TFlex I-cache, so we returned to a more conventional I-cache design, where a block's header and body are split into 32-byte cache lines, and are managed independently by the cache control logic. When a block is fetched, the header line is accessed in the I-cache to find the block's size, at which point the body cache lines (aside from the first, which is fetched with the header) are fetched, decoded, and dispatched to the issue window. If a body line misses, the cache blocks until the miss returns from the shared L2. At that point the instructions of the block are dispatched and can be executed.

Issue Queue. Since multiple blocks reside in the issue queue simultaneously, the issue queue and decode logic must encode and decode queue offsets. This logic tracks and adds the offset of the start of the new block to write instructions into the correct buffers. The wakeup logic requires more complexity. In the single-block case, the target bits that identify a producer's consumer instruction also index to the issue queue. With multiple blocks in flight, the base of each block must be stored and added to the target ID to compute the actual instruction target in the issue window. By restricting block starting points to eight-instruction alignment boundaries, only the high-order four bits of an instruction's target ID must be added to the four-bit number identifying the starting eight-instruction segment of this particular block, reducing this computation to a four-bit addition per intra-block producer/consumer communication.

Parameter	Configuration
Instruction cache	32 KB; 1-cycle hit; 4-way set associative
Branch predictor	Local/Gshare tournament predictor (8K+256 bits, 3-cycle latency). Entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16. BTB: 128, Btype: 256
Data cache	32 KB; 2-cycle hit; 2-way set associative; 1 read/1 write port; 44-entry LSQ
Issue width	Limited dual issue (up to 2 integer and 1 floating-point)
Issue window	Depends on block size
L2 cache	4 MB S-NUCA [9]; 5–27 cycle hit latency depending on address
Main memory	Average unloaded latency: 150 cycles

Table 1. Microarchitectural parameters of each TFlex core

6. Architectural Results

We evaluate the effects of block size and variability using a cycle-level simulator based on TFlex, which we extend to support multiple, variable block sizes and to map multiple blocks per core. We measure performance using one to 32 cores, where each core has the microarchitectural parameters described in Table 1. For each core count, we use a different block size: 32, 64, or 128 instructions. Given a maximum block size the compiler attempts to produce the largest possible blocks. Because we vary the core count and block size, the total instruction window size depends on the number of cores, the block size, and the number of blocks mapped per core.

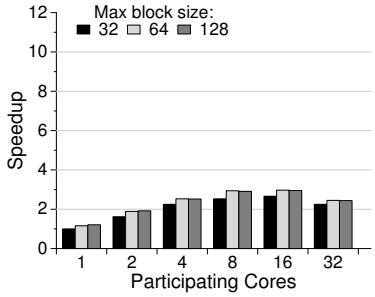
We measure performance on the SPEC CPU2000 benchmarks using early SimPoint methodology [20]. In the results we separate integer from floating point benchmarks, as these suites show significantly different trends. We use these single-threaded workloads, rather than multiprogrammed or multithreaded workloads, to explore the design space of each individual core, the composition of cores, and their interactions for a single thread.

6.1. One Block Per Core

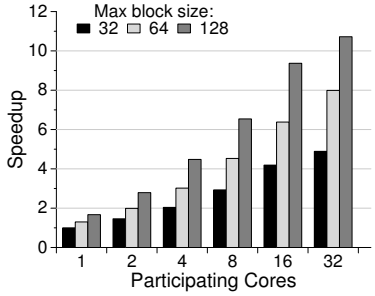
The baseline TFlex design maps one block to each core. To isolate the effect of block size on performance we vary the block size from 32 to 128 instructions while mapping only one block per core. Smaller block sizes thus are at a disadvantage in terms of total window size: the maximum possible instruction window size is the block size multiplied by the number of cores.

Figure 4 shows the geometric mean speedup of the SPECINT and SPECFP benchmarks, normalized to the cycle count of a single TFlex core with support for 32-instruction blocks. The performance trends differ markedly between SPECINT and SPECFP. For all block sizes, floating point performance improves as core count increases, whereas integer performance reaches a maximum between four and eight cores, depending on block size.

On SPECFP, the difference between 32 and 128-

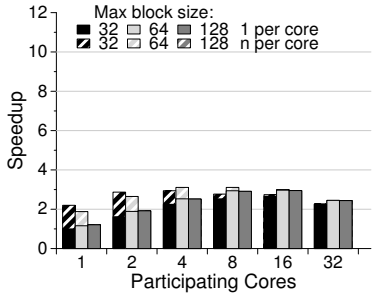


(a) SPECINT

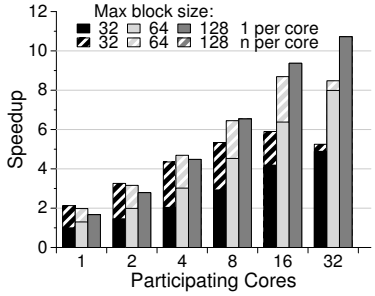


(b) SPECFP

Figure 4. Performance with one fixed-size block per core and maximum fixed block size of 32, 64, and 128 instructions. Fewer hardware resources are required with smaller maximum block sizes.



(a) SPECINT



(b) SPECFP

Figure 5. Performance with 128 instructions per core and fixed-size blocks. Thus, with maximum block sizes of 32, 64, and 128 instructions, there are 4, 2, and 1 blocks per core, respectively, and all configurations require the same number of issue queue slots.

instruction blocks ranges from a factor of 1.6 at one core to a factor of 2.2 at 32 cores. While 32-instruction blocks achieve a speedup of 4.4 at 32-cores on SPECFP, 128-instruction blocks achieve an 11x speedup.

On SPECINT the best-performing block size and core combination is 64 instructions blocks with eight participating cores. The performance characteristics of integer benchmarks are explained by the code characteristics described in Section 4. Blocks in integer benchmarks are frequently small, thus yielding no benefit for large architectural block sizes. When integer benchmark blocks are large, they are typically deeply predicated, which leads to the inclusion of many useless instructions.

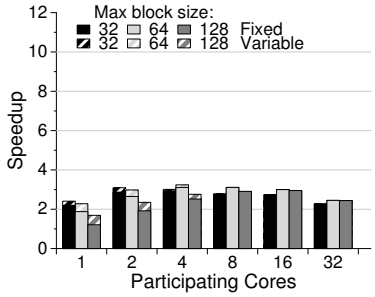
6.2. Multiple Blocks Per Core

Figure 5 shows performance when each core can map multiple blocks of a fixed size. In these experiments the maximum window size per core is fixed at 128 instructions, so the 32-instruction per block configuration executes four blocks concurrently, while the 128-instruction per block configuration executes only one block per core. Compared to one block per core, the increased window size improves performance, particularly at low core counts where the speculation depth is low. The maximum performance on SPECINT always occurs when a maximum of 16 blocks are in flight: four cores with 32-instruction blocks, eight cores with 64, and 16 cores with 128.

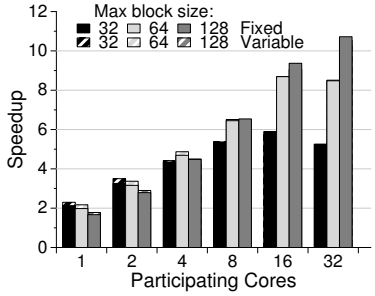
Integer programs tend to have smaller blocks, so allowing more small blocks in flight yields better performance than fewer large blocks. For example, four 32-instruction blocks in flight per core requires the same issue queue space as a single 128-instruction block, but leads to better resource utilization. While supporting more blocks in flight increases complexity, the additional performance may be worth that cost.

SPECINT performance in this configuration saturates quickly, achieving a maximum speedup of 3.1x with four cores and 64-instruction blocks. Even one core with 32-instruction blocks, however, yields a 2.2x speedup. At low core counts, smaller blocks have an advantage because they are less deeply predicated and instead take advantage of branch prediction. Only once the machine has a large issue width does it make sense to use larger blocks.

SPECFP benchmarks perform best with large blocks. Even though these hardware configurations all provide a 128-instruction window per core, larger blocks still outperform smaller blocks, because intra-block communication is much more efficient than inter-block communication. Since SPECFP blocks are generally full of useful instructions with very little predication, there is no downside to using large blocks, and the lower communication latency produces a significant performance win.



(a) SPECINT



(b) SPECFP

Figure 6. Performance with 8-instruction granularity variability in the instruction window with various maximum block sizes using 128 instructions per core.

These results motivate the flexibility of the variable-size blocks architecture. Ideally, a system should be able to achieve the high floating-point performance of 128-instruction blocks, while still achieving high performance on integer benchmarks with smaller block sizes.

6.3. Variable-size Blocks

We measure performance with variable-size blocks at an eight-instruction granularity as described in Section 5, and with a maximum block size of 128-instructions. This fine granularity enables performance improvements over any of the three fixed block sizes evaluated in the previous section. This performance improvement, however, is secondary to the increase in flexibility. By supporting variable-size blocks, a single microarchitecture can achieve both high integer performance using smaller blocks, and high floating-point performance using larger blocks. With a flexible microarchitecture, the compiler must use heuristics to determine the appropriate block size for a benchmark.

Figure 6 shows the performance of this microarchitecture with various maximum block sizes. Variable-size blocks improve performance at small core counts, particularly on SPECINT benchmarks with small blocks. Because the baseline compiler is not aware of variable-size blocks, it creates larger but less efficient blocks when given a larger upper bound. This result motivates an investigation of compiler support for such architectures.

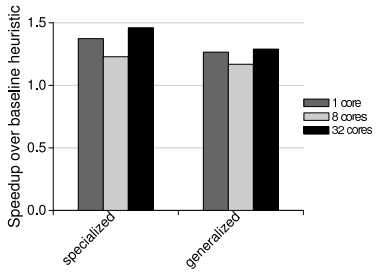


Figure 7. Speedup of learned heuristics on microbenchmarks normalized to the baseline heuristic at the same core count.

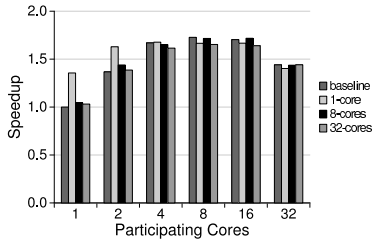
7. Compiler Results

To take advantage of the flexibility provided by variable-size blocks the compiler must apply effective policies. Composability influences the choice of heuristics, as the compiler can perform better if it knows the hardware configuration. To rapidly search the space of heuristics we employ machine learning, and attempt to explain the heuristics that it generates. We select a learning technique based on genetic programming because the result is a heuristic function that can be interpreted easily by humans.

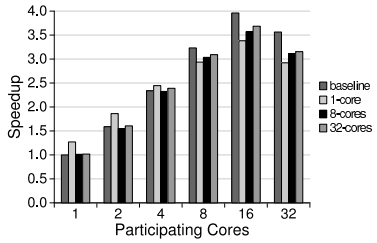
We use meta-optimization, proposed by Stephenson et al. [24], to learn hyperblock formation heuristics. Heuristic functions are N-ary trees of operators, with a mix of code features and constants at the leaves. Each generation consists of 300 heuristics, with the first generation initialized with a combination of hand-written and randomly generated heuristics. We evaluate the performance of each heuristic in a generation and use these scores to determine which organisms will survive to the next generation. To form the next generation, we apply crossover and mutation to introduce variation into the population and probabilistically discover higher performance heuristics [11].

Because simulation of full benchmarks is too time-consuming for learning, we use a group of microbenchmarks for training and then apply the learned heuristics to full benchmarks. We draw these microbenchmarks from SPEC, signal processing, and high-performance computing kernels. We train a group of heuristics on configurations consisting of one, eight, and 32 processors to discover differences in policy for these topologies.

We use a two-phase training methodology, which first learns a specialized heuristic for each benchmark, and uses those learned heuristics as a starting point for learning generalized heuristics. For specialized learning we seed the population with our best hand-tuned heuristic and 299 random heuristics, and use performance on that benchmark as the fitness function. For generalized learning we seed the population with the specialized heuristics plus randomly-generated heuristics, and use geometric mean speedup over the suite as the fitness function. For both specialized and generalized learning we allow 50 generations.



(a) SPECINT



(b) SPECFP

Figure 8. Speedup of learned heuristics for 1, 8, and 32-cores over baseline on one core.

Figure 7 shows the speedup compared to the baseline heuristic using this technique. The "specialized" bar shows the geometric mean performance improvement when each benchmark is compiled with its best heuristic. The "learned geomean" bar of Figure 7 shows the geometric mean speedup achieved by the generalized heuristic. For each core count, we normalize to the performance of the baseline system using the same number of cores, so the bars show only the speedup due to the changed heuristic, rather than also reflecting the difference due to core count.

7.1. Learned Heuristics

The best overall heuristic for a one-core configuration differs significantly from the best heuristic for eight- and 32-core configurations. The 1-core heuristic does not use predication, but only merges blocks with unconditional branches. The 8-core heuristic behaves very similarly to the baseline heuristic described in Section generally favors wide predication with limited tail duplication. The 32-core heuristic demonstrates one of the weaknesses of machine learning as a tool for gaining insight: the function generated by the learner is too complex for us to extract meaningful insight about the policy.

The one-core heuristic is significant because it avoids predication, which is an important feature of an EDGE ISA, and differs in this respect from the best heuristics for larger topologies. A one-core configuration lacks the resources to tolerate more widely predicated code as mispredicated instructions occupy scarce issue and execution slots that could be put to better use. At the same time, the 128-instruction window provided by a single core is small enough for effective branch predictor. Larger configurations, by contrast, suffer from geometrically increasing misprediction rates without the use of predication.

While composability is an effective technique for improving performance of the same binary executable by aggregating cores, this result suggests that the compiler could improve performance by targeting a particular configuration. This capability could be useful to specialized code for power-constrained systems, or in a dynamic optimization system that could compile code differently based on the microarchitectural configuration.

7.2. SPEC CPU Results

Figure 8 shows the performance of the learned heuristics for one, eight, and 32-core configurations applied to the SPEC benchmarks on all core counts, normalized to the performance of a single core with the compiler's baseline heuristic. The one-core heuristic performs well in one-core configurations, outperforming the baseline by 35% on SPECINT and 27% on SPECFP, but tapers off at four or more cores, performing roughly equivalently on INT, and as much as 22% below the baseline on FP.

Results for eight and 32-core topologies achieve similar performance to the baseline on SPECINT, but fall short on SPECFP with eight or more cores. Compared to the eight-core heuristic, the baseline achieves 6% higher performance at eight cores and 14% at 32-cores. This result shows the performance fragility of learned heuristics: despite improvements on microbenchmarks, SPEC performance declined with learned heuristics.

8. Conclusions

EDGE blocks expose fine-grain concurrency within a single block, and medium grain concurrency between blocks. The results show that a single, fixed-size block is too rigid for most programs, given compiler capabilities. Providing variable-size blocks adds architectural complexity but provides the compiler with a better target. With this flexibility, the compiler and processor can achieve improved performance on both floating-point and integer programs.

The flexibility of variable-size blocks opens a new avenue for optimization. By exploiting small blocks and knowledge about the microarchitectural configuration, the compiler can improve performance via judicious use of predication. Small configurations with this optimization should prove attractive in power-constrained systems, where some level of out-of-order performance is desired but low power is imperative.

Integrating hardware and software design will become increasingly necessary due to the slow down of device scaling. This paper contributes an example of increasing a system's performance by simultaneously considering the unit of atomicity together with the needs and capabilities of a system's hardware and software.

Acknowledgments

This work is supported by NSF SHF-0910818, NSF CSR-0917191, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, Intel, IBM, CISCO, Google, Microsoft, and a Microsoft Research fellowship. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS team. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004.

[2] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[3] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[4] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the TRIPS computer system. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2009.

[5] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *MICRO-29: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–200, 1996.

[6] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[7] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1–2):229–248, 1993.

[8] E. Ipek, M. Kirman, N. Kirman, and J. 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, 2007.

[9] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS '02: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[10] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO-40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–394, 2007.

[11] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[12] B. A. Maher. *Atomic Block Formation for Explicit Data Graph Execution Architectures*. PhD thesis, The University of Texas at Austin, 2010.

[13] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *MICRO-39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, 2006.

[14] 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 *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, 1992.

[15] S. W. Melvin and Y. N. Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–296, 1991.

[16] S. W. Melvin and Y. N. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[17] S. W. Melvin, M. C. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *MICRO-21: Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, pages 60–63, 1988.

[18] N. Nethercote, D. Burger, and K. S. Mckinley. Convergent compilation applied to loop unrolling. *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 140–158, 2007.

[19] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *MICRO-41: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 23–34, 2008.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[21] L. T. Simpson. *Value-driven Redundancy Elimination*. PhD thesis, Rice University, 1996.

[22] A. Smith, J. Burrill, J. Gibson, B. A. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, 2006.

[23] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *MICRO-39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–102, 2006.

[24] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90, 2003.