Elastic Threads on Composable Processors

Changkyu Kim Simha Sethumadhavan Nitya Ranganathan Haiming Liu
Robert McDonald Doug Burger Stephen W. Keckler
Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

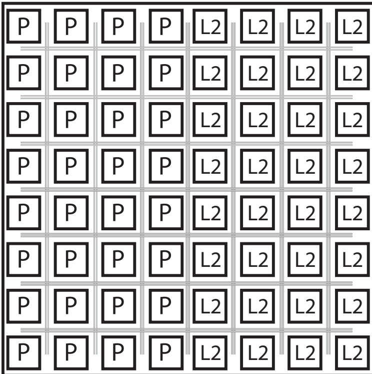
Department of Computer Sciences
Technical Report TR-2006-09
The University of Texas at Austin

August, 2006

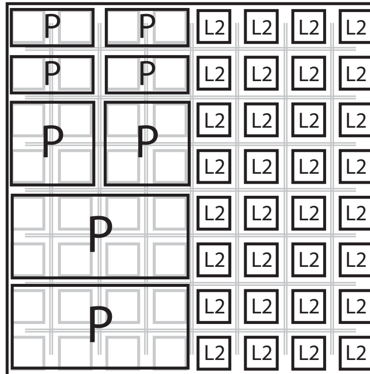
Abstract

Modern CMPs are designed to exploit both instruction-level parallelism within processors and thread-level parallelism across processors, but the balance between the granularity of each processor and the number of processors must be chosen at design time. In this paper, we propose a microarchitecture that allows this balance to be dynamically adjusted. The microarchitecture, which implements the TRIPS instruction set, consists of a large number of fine-grained, single-issue processor cores. By changing a set of OS-visible configuration registers, the system software can aggregate multiple cores to form larger, more powerful processors, depending on the needs of the available threads. For instance, a 64-core chip could be configured as 64 1-wide processors, 1 64-wide processor, or any combination in between. We quantify the area and performance overheads associated with providing the capability to compose larger processors out of multiple small ones, find the distinct ideal configuration for each of several applications, and show the additional benefits gained by explicit compiler support for specific configurations.

(a) 32 1-wide TFlex config.



(b) 8-processor TFlex config.



(c) One 32-wide TFlex config.

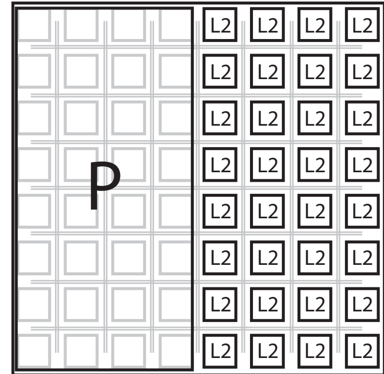


Figure 1: A 32-core, 65nm TFlex chip

1 Introduction

The recent reduction in frequency scaling rates implies that most performance improvements in the future will come from exploiting more concurrency. Concurrency can be exploited by many levels of modern systems: by hardware (ILP/superscalar processors), by support in the ISA and compiler (VLIW architectures), or by the compiler or programmer in parallel systems. Since superscalar and VLIW processors' widths have not scaled recently, industry has been moving to chip multiprocessors in the hope that software threads will provide the concurrency needed for future performance gains.

The disadvantage to CMPs is their relative inflexibility. In current designs, the granularity (i.e. issue width) and number of processors on each chip are fixed at design time, based on the designers' best guesses about workloads and power/performance requirements. Once deployed, however, the right balance between granularity and number of cores per chip changes as the workload mix changes. Currently, simultaneous multithreaded (SMT) cores on a chip multiprocessor are designers' best attempt to provide some flexibility in balancing the number of threads running with the performance of each thread.

In this paper, we describe ISA and microarchitectural support for a composable CMP, that consists of a large number of simple, single-issue processor cores that can be aggregated to form more powerful logical processors. We use the term "core" to denote one of the simple, single-issue processor tiles, and "processor" to denote a logical processor formed from one or more hardware cores. In the design we simulate in this paper, each thread can be run transparently on a logical processor that may be composed of any power-of-two number of cores between one and 32 cores (although 128 is the theoretical maximum) as shown in Figure 1.

The composability is provided on two levels. At the ISA level, we exploit the features of an Explicit Data Graph Execution (EDGE) instruction set. Specifically, we use the TRIPS instruction set, which breaks programs up into sequences of 128-instruction blocks that internally execute in dataflow order. Each core in this microarchitecture has a 128-instruction window, and can thus run one TRIPS block at a time. If two cores are participating, each core can hold half (64 instructions) of each of two blocks, thus doubling the data cache capacity and bandwidth, issue width, and issue window size.

The second level is microarchitectural. To achieve a high degree of composability, the microarchitecture must be fully distributed, with no centralized resources. The composable microarchitecture contains innovations that permit instruction fetch, branch prediction, block control, and memory disambiguation to be fully distributed. All microarchitectural resources are used no matter how many cores are allocated per thread.

This complete distribution allows high flexibility when composing processors. A chip with 32 cores could obviously run 32 threads on one core each (Figure 1a), but the same chip could also run one thread across 32 cores, supporting 32 blocks in flight for a maximum window size of 4096 instructions (Figure 1c), or any point in between. For example, such a chip could also support configurations like one running eight threads: two threads running on eight cores each, two threads running on four cores each, and four threads running on two cores each (Figure 1b).

The capability to balance the number of processors with the granularity of individual processors gives the OS significant flexibility with respect to choosing the hardware's ideal configuration. The right mix depends on a number of factors. With more available threads, if job throughput is paramount, the right approach is to scale down the processor granularity and run threads on fewer cores each. If the system is near power or thermal limits, the running threads may be scaled down to fewer cores each, with the unused cores being clock- or power-gated.

Depending on its available parallelism and execution behavior, each thread will have an ideal number of cores that maximizes performance. For threads with low fine-grained concurrency, or poor cache or predictor behavior, a small number of cores provides the ideal point. For predictable benchmarks with copious fine-grained concurrency, the ideal number of cores can be as many as 16 or 32. Software support by the compiler has the potential to alter this ideal point, since code can be spatially scheduled to be most amenable to running on large, wide-issue logical processors. In this paper, we measure the performance of kernels running on a range of processor sizes (from single-issue to 32-wide) both with and without size-explicit scheduling support from the compiler.

In the rest of this paper, we first describe related work that has provided aspects of processor composability. Section 3 then describes the microarchitectural support needed to compose these processors, including the high-level microarchitecture, distributed control and dependence prediction, distributed disambiguation, and the control register interface. Section 4 evaluates performance and performance potential, quantifying the sources of performance loss in the fully distributed microarchitecture. Section 5 shows initial performance improvements possible if compiler support is added to schedule instructions for specific processor sizes. Finally, we conclude in Section 6.

2 Related approaches

The ability to adapt multiprocessor hardware to fit the needs of the available software is clearly desirable, both in terms of overall performance and power efficiency [3, 11]. The amount of prior research performed to address this problem has been considerable, and falls into three broad categories. In the first, researchers design large cores and provide the capability to resize or share subcomponents of the processor. In the second, researchers attempt to provide higher single-thread performance from a collection of distributed units. In the third, researchers explicitly implement multiple distinct granularities to match up the software with the appropriate hardware.

2.1 Decomposing large cores

The most popular approach for decomposing large cores to date has been Simultaneous Multithreading [22], in which multiple threads share a single large, out-of-order core. The OS achieves adaptive granularity by adjusting the number of threads that are mapped to one processor. The advantages to SMT are extremely low overheads for providing the adaptive granularity. The disadvantages are that the range of granularity is limited, since processors are typically restricted to be four-wide, and threads sharing the same core may cause significant interference.

What Albonesi has termed “adaptive processing” [2] involves dynamically resizing large structures in an out-of-order core, powering fractions of them down based on expected requirements, thus balancing power consumption with performance, by efficiently mapping threads to right-sized hardware structures. Researchers have proposed adjusting cache size via ways [1], issue window size [8], the issue window coupled with the load/store queue and register file [17], and issue width, along with the requisite functional units [4]. While adaptive processing permits improved energy efficiency by adjusting the core’s resources to the needs of the running application, it does not permit a fine-grained tradeoff between core granularity and number of threads. While combining adaptive processing with SMT might achieve that goal, the complexity and overheads on a large-centralized core would be significant

Finally, conjoined-core chip multiprocessing [16] aims to provide some shared resources, with other explicitly partitioned resources, effectively creating a hybrid between SMT and CMP approaches. Similar to SMT approaches, the degree of granularity configuration between single threads versus multiple threads has a range more limited than the approach presented in this paper.

2.2 Composing Processors from Smaller Cores

Many research efforts have aimed to synthesize a more powerful core out of smaller or clustered components. Clustered superscalar processors [5] and the compiler-supported Multicluster design [6] both aim to improve the scalability of a large, out-of-order superscalar by using multiple, clustered execution resources. While this approach improves the complexity of each cluster, it shares the disadvantage that adaptive processing has of the inability to trade off multiple threads for core granularity.

More akin to this paper is the prior work that builds larger logical processors out of smaller discrete units. Most of this prior work uses independent sequencers in the smaller units, even for the modes in which the distributed resources are aggregated for a single software thread. Multiscalar processors [19] used speculation to fill up independent processing elements (called *stages*), with each of the speculative stages starting from a predicted, control-independent point in the program. The Multiscalar design used a shared resource (the ARB) for memory disambiguation, and did not permit the stages to run distinct software threads independently.

The subsequent speculative threads work [10, 14] adapted the Multiscalar execution model to a CMP substrate that could execute separate threads on the individual processors when not in speculative threads mode. This approach is the most similar to that which we describe in this paper, but differs in that each speculative thread has an independent sequencer, thus forming a discontiguous logical instruction issue window, which can create additional mis-speculations and extra communication overhead.

Thread-level speculation, as well as this work, aim for composable processors using out-of-order issue for high ILP. Other composable approaches have provided statically exposed architectures that can be partitioned. The best example is the RAW architecture [21], an important and early tiled architecture. The RAW compiler can target any number of single-issue RAW tiles, forming a single static schedule across them. Each tile still has its own instruction sequencer, although they are highly synchronized with one another. Multiple tasks can be run across a set of tiles provided that each task was compiled for the number of tiles to which it was allocated.

Zhong et al. propose a VLIW architecture whose distributed slots have independent sequencers [24]. While this architecture does not provide multi-grain configurability, it is a small-step to provide RAW-like reconfiguration for multiple threads. The distributed sequencers are related to the distributed control predictor we describe in this paper.

2.3 Providing Multiple Granularities

Finally, some proposals aim to match an application's granularity needs by providing the hardware that best suits the application. Single-ISA, heterogeneous multi-core architectures [15] provide a discrete number of processor sizes on a CMP. This approach increases design complexity and limits the number of granularity options, but does not suffer from the overhead of making the processors variable-grain or composable. Alternatively, custom-fit processors choose the right grain size for specific applications at design time, which clearly increases efficiency at the expensive of run-time flexibility [7, 9].

3 Microarchitectural Support for Composable Processors

Fully composable processors must necessarily be completely distributed, with no centralized hardware structures. In addition, it is desirable to have no unused hardware either in small processor mode or large processor mode. RAW microprocessors [23] achieve this goal, with the disadvantage of requiring the compiler to generate code for one configuration only *a priori*.

The composable microarchitecture we describe here implements the TRIPS ISA [18]. Called TFlex, the microarchitecture supports complete distribution of all necessary hardware structures, with a small amount of additional hardware for large configurations. The TRIPS ISA already supports distributed register files, instruction issue logic, and ALUs, as seen in the original TRIPS microarchitecture. TFlex adds four capabilities in a distributed fashion: I-cache management, next-block prediction, L1 D-cache management, memory disambiguation hardware. We describe the distribution approach for all structures below.

3.1 TFlex core microarchitecture

Figure 2 shows the high-level microarchitecture of a single TFlex core. While many variants are possible, this is the default configuration that we simulate. Each individual core has sufficient hardware resources to run exactly one TRIPS instruction block at a time: 128 reservation stations, a load/store queue containing 40 entries, a 8-Kbit next-block predictor, an 4KB L1 I-cache bank, a 4KB block header cache bank, and an 8KB L1 D-cache bank. The 3-ported register file holds the 128 architectural registers. The execution units are connected to a 64-bit wide operand transfer network, and the backsides of the L1 caches are connected to a 128-bit wide memory transfer network, which services misses to the L2.

Each core is a full-fledged processor, capable of issuing a single instruction per cycle from the 128-instruction window. Instructions may be issued out of order, obeying the intra-block dataflow constraints. The bypass network supports back-to-back execution of dependent instructions. Loads can be stalled in the load-store queue, depending on the results from the dependence predictor.

When running in single-core mode, some of the hardware shown in the core is unused. The register forwarding logic for inter-block communication, the operand network queues and bypasses, and eight of the 40 entries in the LSQ are all extra logic to support multi-block execution across multiple cores. However, this added support is small compared to the overall area of the core.

3.2 Area Estimation

Table 1 shows the area occupied by the various components in a TFlex core. We obtained the area estimates for the 130nm technology from an implementation of a similar core implemented in an 130nm ASIC methodology. Note that a custom implementation would likely have better densities. We obtained the estimates for the other technology nodes by assuming linear scaling of both logic and wiring. In a 65nm process, the architecture shown in Figure 1 would require $86.3mm^2$, and be able to function as a high-end, ultra-wide-issue uniprocessor or a fine-grained, 32-way CMP, all in under 100 mm^2.

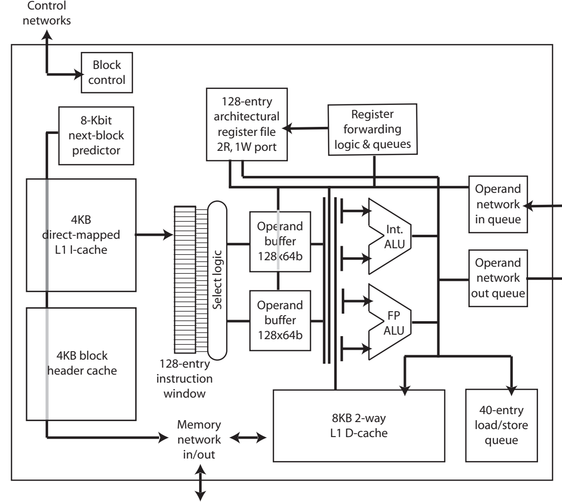


Figure 2: One-core TFlex microarchitecture

Structures per core	Size (in mm^2)			
	130nm	90nm	65nm	45nm
	(impl.)	(scaled)		
Exec. (RS+ALUs+Logic)	2.82	1.41	0.71	0.36
Register Files and Queues	0.87	0.44	0.22	0.11
L1 I-caches	0.92	0.46	0.23	0.12
Next Block Predictor	0.24	0.12	0.06	0.03
L1 D-caches	0.95	0.43	0.22	0.11
Load/Store Queues	0.24	0.12	0.06	0.03
Operand Routers(1x)	0.38	0.19	0.1	0.05
Total Area per core	6.42	3.21	1.61	.81
Total Area for 16 core proc	102.4	51.2	25.6	12.8
Area for 2MB S-NUCA L2	140.5	67.4	35.1	16.9
Area for Figure 1	345.3	169.8	86.3	42.5

Table 1: Area estimates for TFlex cores and CMP

3.3 EDGE ISA background

Explicit Data Graph Execution (EDGE) instruction sets have two distinguishing features. First, they employ *block-atomic execution*, in which groups of instructions execute as a logical atomic unit, either committing all of their output state changes or none (in some sense like transactions). Second, they support *direct-instruction communication* within each block, allowing instructions to specify their dependent instructions in the instruction itself, rather than communicating through a shared namespace like a register file. This feature is what permits the instruction window to be composable; by altering where instruction targets are mapped, blocks may be mapped across a varying number of cores.

The TRIPS ISA is an EDGE architecture that supports 128-instruction blocks, with 32 loads or stores per block. Each target field in each instruction is nine bits; seven bits to specify the consuming instruction number, and two bits to specify whether a produced operand is a left operand, a right operand, or a predicate. Within each block, instructions execute in dataflow order according to the statically generated dataflow graph. Sequential load/store ordering within and across blocks is preserved by five-bit load/store sequence

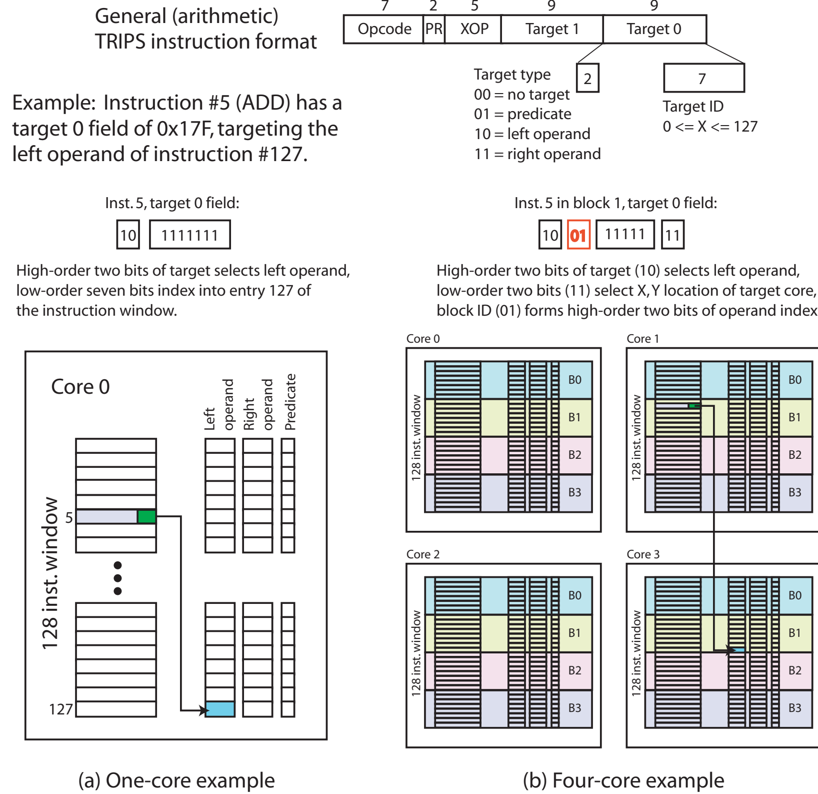


Figure 3: Two-core block mapping

IDs in the instruction bits.

Figure 3 shows how the control logic translates instruction targets to find the dependent instruction that must receive a produced operand. In single-core mode, the translation is trivial: each seven-bit target field merely serves as an index into the 128-entry instruction RAM. The remaining two bits in the target field determine where to save the produced operand. As the number of participating cores grows, more of the 7-bit field is used to determine the destination core, and fewer are used to index into the instruction buffer.

As Figure 3 shows, when running in four core mode, the hardware can support up to four blocks in flight, for a total window size of 512 instructions. Each 128-instruction block is distributed across the four cores, so each instruction buffer at each core holds 32 instructions from each of the four blocks in flight. Two of the seven bits now specify the core to which the result must be communicated, and five of the bits specify which of the 32 instructions for that block the result is targeting at the destination core. Using these features of an EDGE ISA, distributing the instruction issue window and ALUs across multiple cores is straightforward.

3.4 Distributed branch prediction and control

When running a thread across multiple cores, one alternative is to designate one core as the master control core, handling all next-block prediction, fetch commands, misprediction flushing, and block commit. The downside to this alternative is that the branch prediction state and control logic in the other participating cores will go unused.

In the TFlex microarchitecture, we instead implement a distributed control protocol. Each in-flight block is owned by exactly one core, which initiates the fetch, predicts the next block, and commits the block. Block ownership is determined by the low-order bits of the block address. When a block emits its exit branch, the

branch is routed to the owning block (based on its address) and the speculation is confirmed (or a flush is generated).

The distributed next-block predictor uses an Alpha 21264-like local-global tournament predictor with a return address stack. To perform distributed block exit and target prediction, several extensions are necessary. The local history table does not change, since branches that map to a given core will always map to that core, preserving local histories. To support global prediction, the global history register is transmitted from core to core as each prediction is made. Since the prediction tables in each core are very small, we use history folding to support longer histories. On a flush, the core owning the block generating the misprediction initiates the correct fetch and re-sends the rolled back global history vector to the new block owning core. For target prediction, the branch and call target buffers are address partitioned. The most challenging aspect to this distributed predictor is maintaining return address stacks correctly. The RAS is distributed across the cores, which requires keeping extra state to maintain it. The stacks from all the participating cores together form a logical global stack. The call core identifier and the return core identifier need to be kept, so the correct core is accessed when performing RAS predictions. The top two entries of the RAS are also held by the current predicting core. These are then updated if necessary and sent to the next core along with the global histories for the next prediction. This is done so that we incur no additional penalty in fetching the top of the stack when we encounter a return immediately following a return. The two overheads of using all of the distributed predictor state are (1) the potentially long latencies to route the global vector and RAS head from core to core for each prediction, and (2) a potentially higher control misprediction rate. More mispredictions are possible because, although every core sees a complete global history when making a prediction, each branch only updates the second-level global table on one core when committing. This reduces constructive aliasing. History folding also impacts the misprediction rates. To support longer histories along with smaller tables and also enable easy partitioning based on block address, we could use perceptron style predictors [12].

3.5 Distributed instruction fetch

There are two options for flexible caching of blocks across a composable substrate. First, the entire block may be cached at the core that owns it, and be delivered to other cores when the block is fetched. Second, the block may be distributed equally in the I-caches across all participating cores. We choose the latter option to achieve increased bandwidth but, at the cost of some additional complexity. If the blocks are distributed across the I-caches, there are two ways to handle control: all cores can compete for all of the global I-cache, or whether a fraction of each I-cache is "owned" by each participating core, in effect assigning each core some number of sets to manage. An additional design option is whether each I-cache has its own tags, permitting the global I-cache to hold fragments of a block, or whether the I-cache banks act as slave banks to a global master. We assume that a fraction of each I-cache is owned by one of the cores, and that the tagless banks act as slave banks. The I-cache hit rate for this design will be lower than if all cores shared all I-cache sets (due to mapping imbalance), but with a greatly simplified block fetch protocol and I-cache miss handling.

In the TFlex microarchitecture, each participating core manages a fraction of the global I-cache, realized as a fraction of the sets in each I-cache bank. For example, if four cores were participating, each of the cores would be responsible for managing 1/4 of the sets in each of the four I-cache banks.

TRIPS blocks contain two portions: a block header of 128 bytes, and 128 4-byte instructions. The headers contain block mode state, information about stores in the block, and register read and write instructions, which orchestrate the injection of register values into the block and aggregate register writes from the block. Each core contains a 4KB header cache, for holding the headers of the cached blocks that it owns, and a 4KB slave I-cache bank. Associated with the header cache are a set of I-cache tags, which inform the core

which blocks it has cached in its distributed slave I-cache banks.

As another example, assume that 16 cores were participating, and that core 2 receives a next-block prediction. It performs a lookup in its block header cache tags while launching the next-block prediction in parallel and routing the prediction (when complete) to the core that owns the next predicted block. On a hit, the index of the line holding the block would be sent to each of the 16 slave I-cache banks, upon which each of them would load eight instructions from their I-cache bank into their issue window. On a miss, the block owner would send a request to memory for 640 bytes (in the case of a full-sized block), decide in which set the block would reside at each bank, and send the relevant instructions to each I-cache bank when the data returned.

The block owner must also distribute the read and write instructions from the header to the pertinent register files. Registers are interleaved among the cores based on the low-order bits of the register (e.g., in an eight-core configuration, core 0 would hold registers 0, 8, 16, etc., up through 120). Currently we do not exploit the unused space in each register file as cores are added to hold transient register writes (those are held in the register forwarding queues shown in Figure 2), although that is a planned extension.

3.6 Distributed memory disambiguation

Each core has an 8KB, 2-way L1 D-cache bank. When running in single-core mode, each thread would have access to only one such bank. With each core added to the composed processor, however, the bandwidth and capacity scales: each running thread obtains 8KB more of L1 D-cache capacity and an additional memory port. Each bank is addressed based on the low-order bits of the L1 index, thus cache-line interleaving the banks across the cores, as originally proposed by Sohi and Franklin [20].

This cache-line interleaving presents two challenges. First, it is difficult for the compiler to place loads and load consumers near the cache bank, even if the number of cores is known at compile time, as the set of banks to which a load will issue is typically unknown. This fact will result in extra routing latency from core to core. We assume a 15-cycle (plus routing delay), 2MB, 32-bank static NUCA [13] L2 array on the right-hand side of the chip.

Second, the hardware must be able to handle disambiguation for all loads in flight. The load/store queues are address interleaved just as the data caches are, raising the possibility of overflowing one LSQ bank. Since each core has a 40-entry LSQ, in one-core mode, the LSQ cannot overflow, since there is only one block in flight with a maximum of 32 loads and/or stores. However, in the two-core configuration, the window size doubles, to 256 instructions, 64 of which may be loads or stores. If more than 40 of those map to a single core, that LSQ partition will overflow. Prior work has shown that both throttling fetch to prevent overflow and flushing on overflows cause significant performance losses.

In the TFlex microarchitecture, we implement flow control to guarantee forward progress and make flushes extremely rare. We reserve a fraction of each LSQ (4 entries) for the oldest block in flight. If an LSQ bank is full, and a load or store from the oldest block arrives, the pipeline is flushed and the oldest block is run in single-block mode to guarantee forward progress. If a load or store from one of the speculative blocks arrives at a core its LSQ bank is full (except for the reserved 4 entries that it cannot use), the request is NACKed, returned to the issue window, where it waits until a block commits before re-issuing. An extra bit per issue window entry is required to track this state. With the 40 entries per LSQ bank and the NACK capability, the performance of the load/store queues is nearly identical to that of fully-sized LSQ banks in every core.

3.7 Configuration interface

When the OS wishes to allocate a group of cores to serve as a single logical processor, it must follow several steps. First, any threads running on those cores must be interrupted and saved in the process table. Second,

the data caches must be flushed, writing back dirty data and invalidating all blocks. Third, the OS must write into configuration registers in the control tiles at each core three pieces of information: (1) how many cores are participating in this processor, (2) what the topology of the logical processor is (i.e. four by four cores), and (3) each processor's location in that topology. With those three pieces of information, the control logic can correctly configure the instruction cache, plus determine to where each register file access, load, store, branch prediction, and operand must be routed by the micronetwork. It is certainly possible for the OS to vary the logical processor size to exploit different application phases, but the phases must be coarse grained to amortize the cost of interrupting the running threads and flushing the caches.

4 Methodology and Results

In this section, we study application scalability and TFlex microarchitecture scalability. We analyze the performance and area overheads of the proposed microarchitecture, and quantify the performance differentials between perfect, centralized and distributed implementations of the branch predictor, LSQs, and I-caches. We also show speedups of the TFlex microarchitecture compared to a Alpha 21264 processor.

The benchmarks are drawn from two different benchmark suites: EEMBC 2.0, and a suite of signal processing kernels from MIT Lincoln Labs. We use all benchmarks except cjpeg and djpeg from the EEMBC suite and the codes are compiled with Scale compiler with O4 options and optimized for 16 cores. We also use hand coded LL kernels (corner turn, convolution, matrix mul, vector add, genetic algorithm). The results are obtained using a cycle-accurate execution driven simulator that executes TRIPS binaries.

4.1 Potential for Composability

One of the rationales for composability is that applications have varied amounts of concurrency. To quantify the variance, we measured the peak theoretical speedup achievable on various applications in the EEMBC and Kernels benchmark suite. Our baseline comparison point was a Alpha 21264 processor, with kernels and EEMBC compiled using the native GEM compiler with the "-O4 -arch ev6" flags set. We chose the Alpha because it has an aggressive ILP core, an ISA that lends itself to efficient execution, and a mature and stable compiler that generates very high-quality code. To generate the number of cycles, we use a validated Alpha 21264 simulator.

The ideal speedups are presented in Figure 4. We computed speedups by executing TRIPS ISA binaries optimized for 16 cores on the TFlex simulator assuming perfect speculation accuracies, 1-cycle execution latency for all instructions including memory, and assuming only operand transfer latency (not contention) between producers and consumers. The graph shows that speedups can range from 1 to 6 for the EEMBC applications and to 14 for the hand optimized binaries. Such wide variance in applications even within EEMBC hints at the benefits possible from a composable architecture.

The difference between the hand-coded kernels and compiled benchmarks is quite large. We note that the TRIPS compiler, while stable and improving gradually, generates code of significantly lower quality than the Alpha native compiler. The difference between hand-tuned code and compiled code can be seen by comparing the two versions of a2time01 and bezier01, two EEMBC benchmarks that we hand-coded and also compiled as a reference. Our hope is that the compiler performance will eventually fall closer to the hand versions than to its current performance, but where between the two points it falls is currently unknown.

In figure 4, the right bar for each benchmark shows the actual speedup of TFlex over Alpha. The speedup number in kernels ranges from 2.1 to 6.4 except for **sha**, which is almost an entirely serial benchmark. While the overall average speedup over Alpha in kernels reaches 460%, EEMBC has the average 6% degradation compared to Alpha.

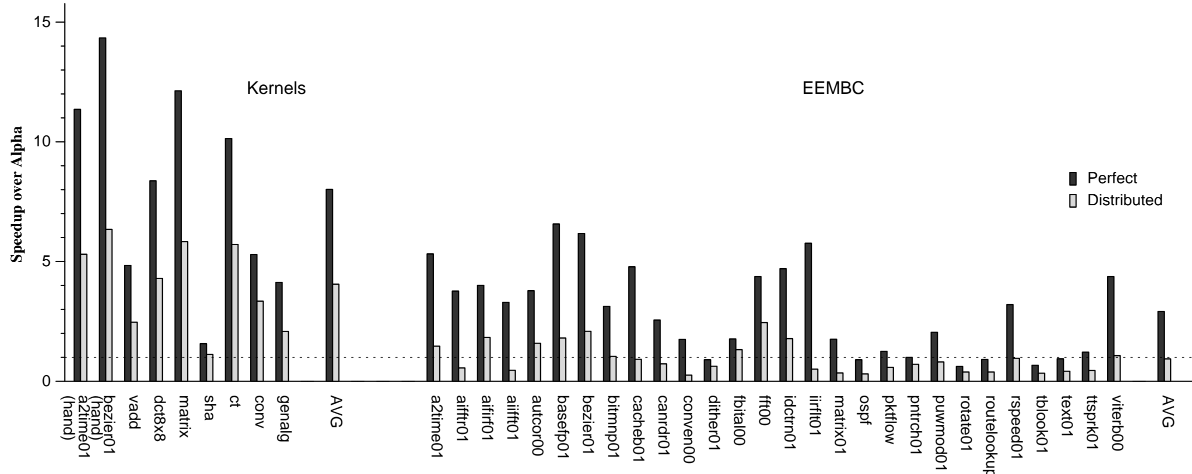


Figure 4: Performance for various applications for processors composed of 1 to 32 cores

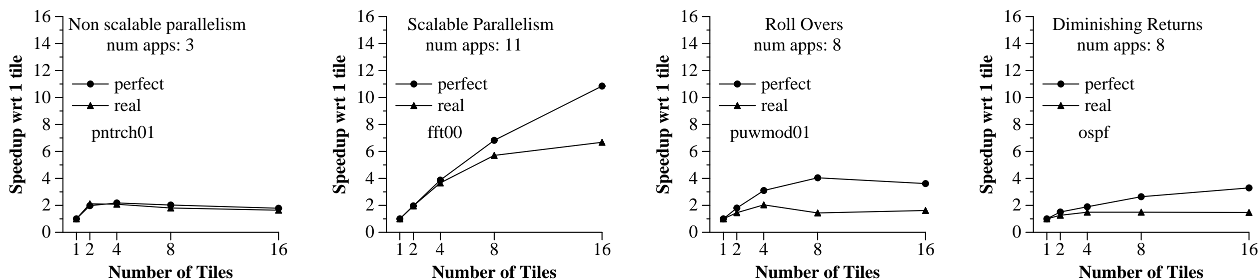


Figure 5: Application scalability of some EEMBC benchmarks 1- 16 cores

Another rationale for composability is that applications have different optimal operating points. In our studies on peak parallelism, we observed four types of application behaviors, illustrated in Figure 5. The figure shows the speedup relative to one core in processors composed of 1 to 16 cores. For applications like pntrch01 (pointer chasing) there is very little inherent parallelism so the performance does not scale as the number of processors increase. For several of the signal processing applications in the EEMBC suite, like fft0, performance scales linearly with an increasing number of cores. For some applications performance peaks over after initial scaling (e.g. puwmod01). For still others applications although the performance scales with number of processors, the performance diminishes with increasing execution resources (e.g. ospf). We note that the flat behavior of some of these codes is simply due to poor code generation by our in-house TRIPS compiler. Nonetheless, depending on the operating conditions and the application scalability the OS may decide to allocate resources to exploit varying degrees of parallelism.

4.2 Microarchitecture Scalability Analysis

The TFlex microarchitecture can perform poorly because of high operand communication latency, due to network congestion, lower branch prediction accuracy, flushes due to LSQ undersizing, and lower I-cache hit rates. In this section, we analyze how each of the above factors affects performance and compare the performance with respect to an idealized implementation.

Operand Delivery: The distribution of execution resources and program partitioning among a large number of cores in TFlex places higher demands on the operand networks compared to other architectures. For

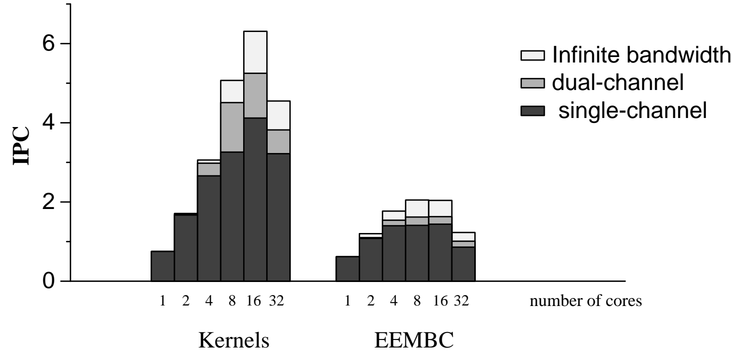


Figure 6: Performance differentials between the infinite bandwidth, dual channel, and single channel network

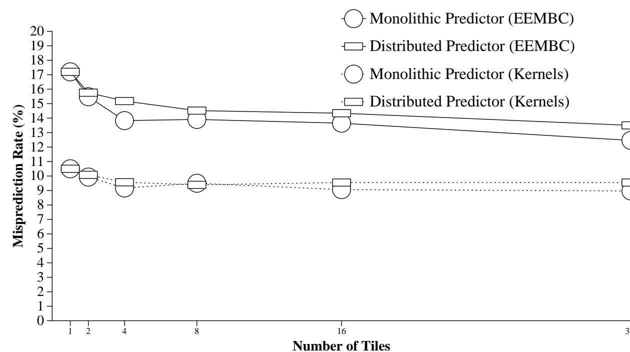


Figure 7: Misprediction rates for monolithic and distributed predictors

instance, significant performance improvements are possible for several applications (as shown in Figure 6) with an infinite bandwidth network compared to the baseline TFlex micronetwork. To improve the performance of the network we can increase the the bandwidth by increasing the number of wires/channels or employ more sophisticated compiler-support instruction placement. In this section we study the former approach while the latter is presented in Section 5. As shown in Figure 6 (even) a dual channel operand network improves the performance by 22% in kernels which have high concurrency and generate heavy operand network traffic. For the rest of the performance analysis section we assume a dual-channel operand network as the baseline.

Distributed Block Predictor Distributing the branch predictor can incur performance losses for several reasons: longer, compressed and folded histories increasing aliasing, loss of constructive aliasing between blocks mapped to different cores, and finally micronetwork contention that might delay the history and RAS top-of-stack updates sent to other cores. However, a distributed predictor also has several potential advantages, such as faster, smaller tables and reduced destructive aliasing. In figure 7 we compare the misprediction rates of distributed and centralized block predictors as we scale the number of cores. The monolithic predictor compared at each core configuration is of the same size as the corresponding distributed predictor (1KB per core). We show the average prediction numbers for each of the two benchmark suites. In general, the distributed predictor performs almost as well as an equal sized monolithic predictor. As we scale the number of cores and give more predictor space we get lower misprediction rates. For the kernels, the mispredcition rate flattens out after 16 cores. This is due to the fact that the kernels have a small code footprint and have very little destructive aliasing. Figure 8 shows the performance in IPC with 3 different configurations: perfect, monolithic and distributed. On the average the performance difference between

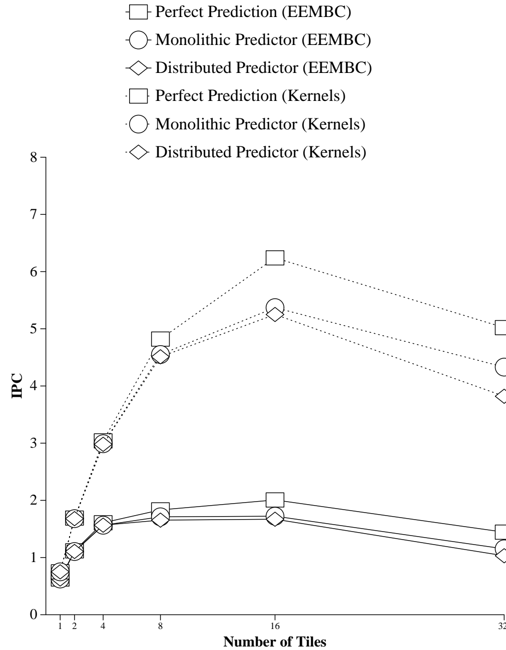


Figure 8: Performance with perfect, monolithic and distributed predictors

monolithic and distributed is very small.

Distributed Memory Disambiguation Distributing the memory disambiguation hardware can result in performance losses in two ways: first, when the instructions from older blocks cannot be slotted in the LSQ, all blocks in execution must be flushed. Second, instructions from younger blocks may have to be replayed, causing additional congestion in the network. Figure 9 reports the performance difference between a two cycle, 40-entry LSQ at each tile (square markers) compared to a two cycle, 512-entry (maximally) sized LSQ replicated at each tile (circle markers). Surprisingly, the distributed implementation sometimes even outperforms the unrealizable maximally sized LSQ. The performance gains come from fewer load violations with the undersized LSQ. In the undersized LSQ, some loads that result in violations in the maximally sized LSQs are delayed (replayed) because LSQ structural hazards and thus resulting in fewer violations. To filter out the effects of memory disambiguation we show the performance results with perfect memory disambiguation for the EEMBC benchmarks. With perfect disambiguation as expected, the undersized LSQ slightly underperforms or closely matches the maximal configuration.

Distributed Cache Figure 10 shows the effect of instruction cache and data cache on the performance of different sized configurations. In the graph, for each group of applications, we show the performance of the applications on different sized configurations with both perfect L1 cache and realistic L1 and L2 cache. As we can see from the graph, the performance curves of the realistic L1 and L2 cache follow the corresponding curve with perfect L1 and L2 cache. All the applications shown in the graph have high L1 instruction cache hit rate across different sized configurations. Therefore, the performance gap between the perfect memory and realistic memory is mainly caused by the L1 data cache. Increasing the number of cores has 2 opposite effects on the L1 data cache performance. On one hand, with more cores, the total capacity of the L1 data cache increases, which will result in higher hit rate. On the other hand, when we increase the number of cores, the average hit and miss latency of the L1 data cache also increases because more cores mean bigger routing distance between the L1 data cache bank and the consumer tile.

In this section, we showed that despite all the theoretical possible losses due to distribution the TFlex microarchitecture performs very close to centralized implementation in most cases.

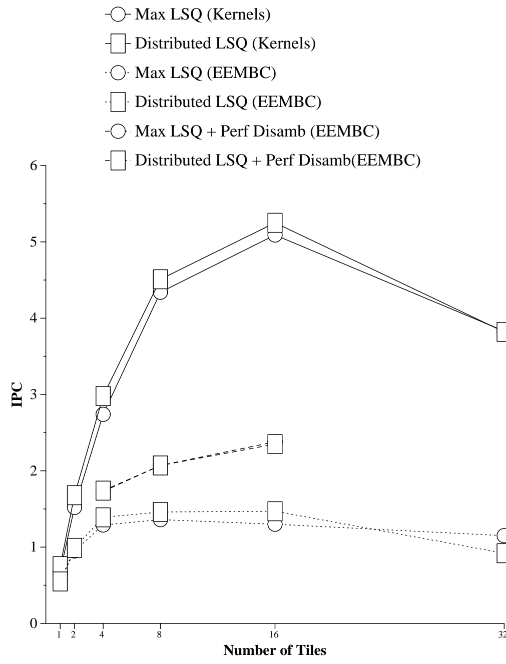


Figure 9: Performance differentials between undersized and Max Sized LSQs

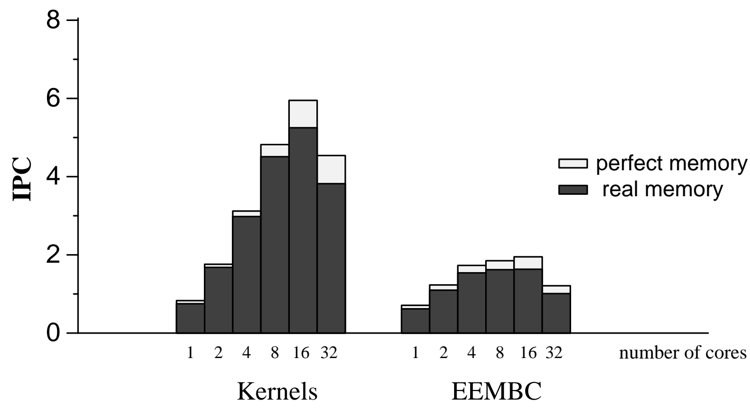


Figure 10: Performance differentials between perfect and real memory

4.3 Area Overheads

In this section, we quantify the area overheads associated with the various microarchitectures. The performance of the TFlex architecture can be improved by increasing the operand bandwidth and upsizing the microarchitecture structures. For instance, doubling the operand bandwidth reduces network contention significantly, closely matching the performance that can be obtained using an infinite network. But the performance comes at the cost of increased area from replicating the operand routers and the associated wires. With the dual channel optimization, the area of the operand network increases by 12.16 mm^2 at 130nm (20% increase in processor area). While, the area overhead of the dual channel network seems high at current technology, transistor scaling will enable such networks in smaller technologies.

Simultaneous performance improvements and area savings are also possible with the TFlex microarchitecture. For instance in the LSQs, a replicated copy of the maximally sized LSQ in each tile will take a whopping 36 mm^2–close to 10% of the die area. The undersized LSQs with 40 entries takes up only (approx) 1% area at 130nm. Additional, but minor, performance savings are possible with a slightly larger 64-entry LSQ (data not presented) but increasing the LSQ can increase the cycle time negatively. Although we assumed fixed sizes for the I-caches, D-caches and branch predictors in this study we plan to explore area-performance tradeoffs for all these parameters in future work.

5 Compiler Support for Composable Processors

For composable processors, the compiler should carefully optimize for communication locality by placing dependent instructions as closely together as possible, achieve good communication/computation load balance by partitioning the program among all resources and provide support for running a binary on multiple hardware configurations. To achieve these goals, the compiler used in this study guesses the critical path within a block and attempts to place instructions in that path as close as possible. However, critical paths cannot be always predicted correctly when the binary is run on different hardware configurations.

While the compilation heuristics (like critical path scheduling) may not be portable across different configurations, the hardware and compiler can mitigate some of reconfiguration effects by using transformations that preserve the communication locality in the original binary. For instance, running the binaries on a smaller-than-scheduled configuration (shrinking) can compensate for some of the losses by reducing the communication distances across all dependent instructions. Running the binaries on a larger-than-scheduled configuration (Dilating) may also be beneficial by reducing critical resource contention (like spreading out FP divides).

To study the performance effects of imprecise compilation i.e. mismatches between the compiled configuration and runtime configuration we used binaries compiled for 1-, 2-, 4-, 8-, 16-, 32- core configurations and ran them on 1-, 2-, 4-, 8-, 16-, 32- core configurations (total 36 possible permutations between binaries and hardware core configurations) assuming only one program was running on the chip at any time. For non matching configurations, we first collapsed adjacent columns towards the L2 memory banks (refer to Figure 1) and then collapsed adjacent rows. Similarly for the dilated binaries we moved instructions in the odd numbered slots to slots in the adjacent cores in the east and southern directions. These transformations are likely to preserve the communication locality assumed or deduced by the compiler in the originally compiler configurations. Tables 2 shows the performance matrix for the all the configurations from kernels

As expected the matching binaries produce the best performance uniformly across all the processor sizes and as the degree of mismatch increases the performance losses grow worse. The 16-core compiled code degrades the performance by 28% when running on the 2-core processor.

Interestingly, the performance loss is less when a binary is compiled at larger granularity and shrunk compared to when it is compiled at smaller granularity and expanded. For example, while there is no

Hardware	Compile Size					
	1-core	2-core	4-core	8-core	16-core	32-core
1-core	0.8	0.8	0.8	0.8	0.8	0.7
2-core	1.7	1.7	1.7	1.7	1.7	1.6
4-core	2.8	2.9	3.0	2.9	2.8	2.8
8-core	3.5	3.5	3.5	4.5	4.4	4.3
16-core	3.8	3.8	3.8	4.7	5.2	4.9
32-core	3.3	3.1	3.1	3.9	3.9	4.9

Table 2: IPC numbers from kernels

average performance loss to run 32-core compiled binaries in 16-cores for TFlex configuration, running 16 core compiled binaries in 32 cores degrades the performance by 25%.

Besides, compared to the best TFlex configuration (using 16-core with 16-core compiled binary), the additional 8% performance improvement can be obtained by customizing the number of cores and binaries to each application.

In summary, the best policy is to profile each application and find out the maximum number of core to run and use the binary compiled for that number of core. If the size of logical processor should be shrunk due to power-saving and multi-thread running requirement, the hardware can minimize the performance loss by compacting adjacent columns and rows to optimize the operand communications

6 Conclusions

In this paper we have described microarchitectural support for run-time configuration of fine-grained CMP processors, allowing great flexibility in aggregating cores together to form larger logical processors. A disadvantage of this approach is that it relies on non-traditional ISA support, using EDGE architectures rather than RISC or CISC. An advantage is that unlike prior work, the larger logical processor groups together distributed resources to form unified logical resources, including instruction sequencing, memory disambiguation, data caches, instruction caches, register files, and branch predictors. That grouping permits higher performance than previous distributed approaches (such as thread-level speculation) as well as a finer degree of configurability.

Since most future performance gains will come from concurrency, future systems will need to mine concurrency from all levels. Depending on the workload mix and number of available threads, the right place to find the concurrency will likely change frequently for general-purpose systems, rendering the design-time freezing of processor granularity in traditional CMPs a highly undesirable option. Composable processors permits the run-time system to make informed decisions about how to go about exploiting concurrency, whether it be from a single thread running on many distributed cores, or many threads running on partitioned resources. Other factors that may affect the resource configuration include power/performance tradeoffs and the amount of concurrency within each thread.

This configurability provides the OS with another degree of freedom when balancing power and performance. In addition to the potential power advantages described earlier, the OS may combine dynamic voltage and frequency scaling with processor composition to optimize performance per watt. For example, depending on the application it may be preferable to run a thread on four cores at 1 GHz or on two cores at 2 GHz.

Interesting directions for the future also involve programming methodologies to expose more concurrency in single threads, better compiler technology for aligning memory operations to individual spatial

banks, and hybrid mechanisms that allow for highly efficient spatial inter-thread communication on these composable substrates.

References

[1] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 248–259, Dec. 1999.

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

[3] M. Annavaram, E. Grochowski, and J. P. Shen. Mitigating amdahl's law through epi throttling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 298–309, 2005.

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

[5] R. Canal, J.-M. Parcerisa, and A. González. A cost-effective clustered architecture. In *Proceedings of the 8th International Symposium on Parallel Architectures and Compilation Techniques*, pages 160–168, 1999.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, December 1997.

[7] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 324–335, December 1996.

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

[9] R. E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, /2000.

[10] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. K. Chen, and K. Olukotun. The stanford hydra cmp. *IEEE Micro*, 20(2):71–84, 2000.

[11] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future CMPs. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, September 2001.

[12] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 197–206, 2001.

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

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

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, 2003.

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

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

[18] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proc. of the 30th Intl Symp on Computer Architecture*, pages 422–433, June 2003.

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

[20] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the 4th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Sept. 1991.

[21] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA*, pages 2–13, 2004.

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

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

[24] H. Zhong, K. Fan, S. A. Mahlke, and M. S. Schlansker. A distributed control path architecture for vliw processors. In *Proceedings of the 14th International Symposium on Parallel Architectures and Compilation Techniques*, pages 197–206, 2005.