Appears in the 30^{th} *Annual International Symposium on Computer Architecture*

Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture

Karthikeyan Sankaralingam Ramadass Nagarajan Haiming Liu Changkyu Kim
Jaehyuk Huh Doug Burger Stephen W. Keckler Charles R. Moore

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

Abstract

This paper describes the **polymorphous** *TRIPS architecture which can be configured for different granularities and types of parallelism. TRIPS contains mechanisms that enable the processing cores and the on-chip memory system to be configured and combined in different modes for instruction, data, or thread-level parallelism. To adapt to small and large-grain concurrency, the TRIPS architecture contains four out-of-order, 16-wide-issue Grid Processor cores, which can be partitioned when easily extractable fine-grained parallelism exists. This approach to polymorphism provides better performance across a wide range of application types than an approach in which many small processors are aggregated to run workloads with irregular parallelism. Our results show that high performance can be obtained in each of the three modes–ILP, TLP, and DLP–demonstrating the viability of the polymorphous coarse-grained approach for future microprocessors.*

1 Introduction

General-purpose microprocessors owe their success to their ability to run many diverse workloads well. Today, many application-specific processors, such as desktop, network, server, scientific, graphics, and digital signal processors have been constructed to match the particular parallelism characteristics of their application domains. Building processors that are not only general purpose for single-threaded programs but for many types of concurrency as well would provide substantive benefits in terms of system flexibility as well as reduced design and mask costs.

Unfortunately, design trends are applying pressure in the opposite direction: toward designs that are *more* specialized, not less. This performance *fragility*, in which applications incur large swings in performance based on how well they map to a given design, is the result of the combination of two trends: the diversification of workloads (media, streaming, network, desktop) and the emergence of chip multiprocessors (CMPs), for which the number and granularity of processors is fixed at processor design time.

One strategy for combating processor fragility is to build a heterogeneous chip, which contains multiple processing cores, each designed to run a distinct class of workloads effectively. The proposed Tarantula processor is one such example of integrated heterogeneity [8]. The two major downsides to this approach are (1) increased hardware complexity since there is little design reuse between the two types of processors and (2) poor resource utilization when the application mix contains a balance different than that ideally suited to the underlying heterogeneous hardware.

An alternative approach to designing an integrated solution using multiple heterogeneous processors is to build one or more homogeneous processors on a die, which mitigates the aforementioned complexity problem. When an application maps well onto the homogeneous substrate, the utilization problem is solved, as the application is not limited to one of several heterogeneous processors. To solve the fragility problem, however, the homogeneous hardware must be able to run a wide range of application classes effectively. We define this architectural *polymorphism* as the capability to configure hardware for efficient execution across broad classes of applications.

A key question, is what granularity of processors and memories on a CMP is best for polymorphous capabilities. Should future billion-transistor chips contain thousands of fine-grain processing elements (PEs) or far fewer extremely coarse-grain processors? The success or failure of polymorphous capabilities will have a strong effect on the answer to these questions. Figure 1 shows a range of points in the spectrum of PE granularities that are possible for a $400mm^2$ chip in 100nm technology. Although other possible topologies certainly exist, the five shown in the diagram represent a good cross-section of the overall space:

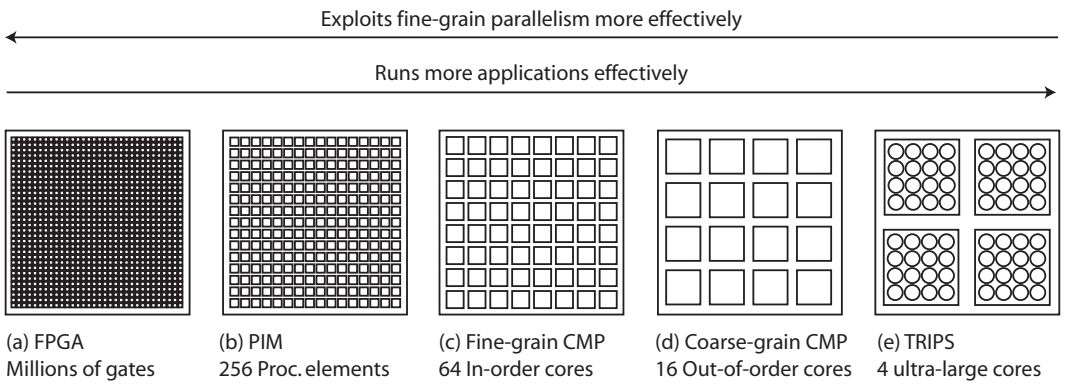


Figure 1. Granularity of parallel processing elements on a chip.

a) Ultra-fine-grained FPGAs.

b) Hundreds of primitive processors connected to memory banks such as a processor-in-memory (PIM) architecture or reconfigurable ALU arrays such as RaPiD [7], Piperench [9], or PACT [3].

c) Tens of simple in-order processors, such as in RAW [25] or Piranha [2] architectures.

d) Coarse grained architectures consisting of 10-20 4-issue cores, such as the Power4 [22], Cyclops [4], MultiScalar processors [19], other proposed speculatively-threaded CMPs [6, 20], and the polymorphous Smart Memories [15] architecture.

e) Wide-issue processors with many ALUs each, such as Grid Processors [16].

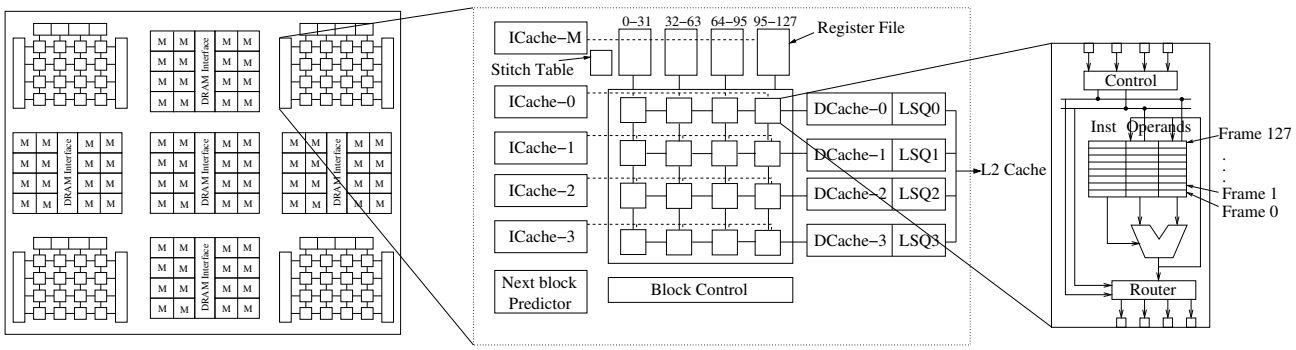
The finer-grained architectures on the left of this spectrum can offer high performance on applications with fine-grained (data) parallelism, but will have difficulty achieving good performance on general-purpose and serial applications. For example, a PIM topology has high peak performance, but its performance on on control-bound codes with irregular memory accesses, such as compression or compilation, would be dismal at best. At the other extreme, coarser-grained architectures traditionally have not had the capability to use internal hardware to show high performance on fine-grained, highly parallel applications.

Polymorphism can bridge this dichotomy with either of two competing approaches. A *synthesis* approach uses a fine-grained CMP to exploit applications with fine-grained, regular parallelism, and tackles irregular, coarser-grain parallelism by synthesizing multiple processing elements into larger "logical" processors. This approach builds hardware more to the left on the spectrum in Figure 1 and emulates hardware farther to the right. A *partitioning* approach implements a coarse-grained CMP in hardware, and logically partitions the large processors to exploit finer-grain parallelism when it exists.

Regardless of the approach, a polymorphous architecture will not outperform custom hardware meant for a given application, such as graphics processing. However, a successful polymorphous system should run well across many application classes, ideally running with only small performance degradations compared to the performance of customized solutions for each application.

This paper proposes and describes the polymorphous TRIPS architecture, which uses the partitioning approach, combining coarse-grained polymorphous Grid Processor cores with an adaptive, polymorphous on-chip memory system. Our goal is to design cores that are both as large and as few as possible, providing maximal single-thread performance, while remaining partitionable to exploit fine-grained parallelism. Our results demonstrate that this partitioning approach solves the fragility problem by using polymorphous mechanisms to yield high performance for both coarse and fine-grained concurrent applications. To be successful, the competing approach of synthesizing coarser-grain processors from fine-grained components must overcome the challenges of distributed control, long interaction latencies, and synchronization overheads.

The rest of this paper describes the polymorphous hardware and configurations used to exploit different types of parallelism across a broad spectrum of application types. Section 2 describes both the planned TRIPS silicon prototype and its polymorphous hardware resources, which permit flexible execution over highly variable application domains. These resources support three modes of execution that we call *major morphs*, each of which is well suited for a different type of parallelism: instruction-level parallelism with the desktop or D-morph (Section 3), thread-level parallelism with the threaded or T-morph (Section 4), and data-level parallelism with the streaming or S-morph (Section 5). Section 6 shows how performance increases in the three morphs as each TRIPS core is scaled from a 16-wide up to an even coarser-grain, 64-wide issue processor. We conclude in Section 7 that by building large, partition-



(a) TRIPS Chip (b) TRIPS Core (c) Execution Node

Figure 2. TRIPS architecture overview.

able, polymorphous cores, a single homogeneous design can exploit many classes of concurrency, making this approach promising for solving the emerging challenge of processor fragility.

2 The TRIPS Architecture

The TRIPS architecture uses large, coarse-grained processing cores to achieve high performance on single-threaded applications with high ILP, and augments them with polymorphous features that enable the core to be subdivided for explicitly concurrent applications at different granularities. Contrary to conventional large-core designs with centralized components that are difficult to scale, the TRIPS architecture is heavily partitioned to avoid large centralized structures and long wire runs. These partitioned computation and memory elements are connected by point-to-point communication channels that are exposed to software schedulers for optimization.

The key challenge in defining the polymorphous features is balancing their appropriate granularity so that workloads involving different levels of ILP, TLP and DLP can maximize their use of the available resources, and at the same time avoid escalating complexity and non-scalable structures. The TRIPS system employs coarse-grained polymorphous features, at the level of memory banks and instruction storage, to minimize both software and hardware complexity and configuration overheads. The remainder of this section describes the high level architecture of the TRIPS system, and highlights the polymorphous resources used to construct the D, T, and S-morphs described in Sections 3–5.

2.1 Core Execution Model

The TRIPS architecture is fundamentally *block oriented*. In all modes of operation, programs compiled for TRIPS are partitioned into large blocks of instructions with a single entry point, no internal loops, and possibly multiple possible exit points as found in hyperblocks [14]. For instruction and thread level parallel programs, blocks commit atomically and interrupts are *block precise*, meaning that they are handled only at block boundaries. For all modes of execution, the compiler is responsible for statically scheduling each block of instructions onto the computational engine such that inter-instruction dependences are explicit. Each block has a static set of state inputs, and a potentially variable set of state outputs that depends upon the exit point from the block. At runtime, the basic operational flow of the processor includes fetching a block from memory, loading it into the computational engine, executing it to completion, committing its results to the persistent architectural state if necessary, and then proceeding to the next block.

2.2 Architectural Overview

Figure 2a shows a diagram of the TRIPS architecture that will be implemented in a prototype chip. While the architecture is scalable to both larger dimensions and high clock rates due to both the partitioned structures and short point-to-point wiring connections, the TRIPS prototype chip will consist of four polymorphous 16-wide cores, an array of 32KB memory tiles connected by a routed network, and a set of distributed memory controllers with channels to external memory. The prototype chip will be built using a 100nm process and is targeted for completion in 2005.

Figure 2b shows an expanded view of a TRIPS core and the primary memory system. The TRIPS core is an example of the Grid Processor family of designs [16], which are typically composed of an array of homogeneous execution nodes, each containing an integer ALU, a floating point unit, a set of reservation stations, and router connections at the input and output. Each reservation station has stor-

age for an instruction and two source operands. When a reservation station contains a valid instruction and a pair of valid operands, the node can select the instruction for execution. After execution, the node can forward the result to any of the operand slots in local or remote reservation stations within the ALU array. The nodes are directly connected to their nearest neighbors, but the routing network can deliver results to any node in the array.

The banked instruction cache on the left couples one bank per row, with an additional instruction cache bank to issue fetches to values from registers for injection into the ALU array. The banked register file above the ALU array holds a portion of the architectural state. To the right of the execution nodes are a set of banked level-1 data caches, which can be accessed by any ALU through the local grid routing network. Below the ALU array is the block control logic that is responsible for sequencing block execution and selecting the next block. The backside of the L1 caches are connected to secondary memory tiles through the chip-wide two-dimensional interconnection network. The switched network provides a robust and scalable connection to a large number of tiles, using less wiring than conventional dedicated channels between these components.

The TRIPS architecture contains three main types of resources. First, the hardcoded, non-polymorphous resources operate in the same manner, and present the same view of internal state in all modes of operation. Some examples include the execution units within the nodes, the interconnect fabric between the nodes, and the L1 instruction cache banks. In the second type, polymorphous resources are used in all modes of operation, but can be configured to operate differently depending on the mode. The third type are the resources that are not required for all modes and can be disabled when not in use for a given mode.

2.3 Polymorphous Resources

Frame Space: As shown in Figure 2c, each execution node contains a set of reservation stations. Reservation stations with the same index across all of the nodes combine to form a physical *frame*. For example, combining the first slot for all nodes in the grid forms frame 0. The *frame space*, or collection of frames, is a polymorphous resource in TRIPS, as it is managed differently by different modes to support efficient execution of alternate forms of parallelism.

Register File Banks: Although the programming model of each execution mode sees essentially the same number of architecturally visible registers, the hardware substrate provides many more. The extra copies can be used in different ways, such as for speculation or multithreading, depending on the mode of operation.

Block Sequencing Controls: The block sequencing controls determine when a block has completed execution, when a block should be deallocated from the frame space, and which block should be loaded next into the free frame space. To implement different modes of operation, a range of policies can govern these actions. The deallocation logic may be configured to allow a block to execute more than once, as is useful in streaming applications in which the same inner loop is applied to multiple data elements. The next block selector can be configured to limit the speculation, and to prioritize between multiple concurrently executing threads useful for multithreaded parallel programs.

Memory Tiles: The TRIPS Memory tiles can be configured to behave as NUCA style L2 cache banks [12], scratchpad memory, synchronization buffers for producer/consumer communication. In addition, the memory tiles closest to each processor present a special high bandwidth interface that further optimizes their use as stream register files.

3 D-morph: Instruction-Level Parallelism

The *desktop morph*, or D-morph, of the TRIPS processor uses the polymorphous capabilities of the processor to run single-threaded codes efficiently by exploiting instruction-level parallelism. The TRIPS processor core is an instantiation of the Grid Processor family of architectures, and as such has similarities to previous work [16], but with some important differences as described in this section.

To achieve high ILP, the D-morph configuration treats the instruction buffers in the processor core as a large, distributed, instruction issue window, which uses the TRIPS ISA to enable out-of-order execution while avoiding the associative issue window lookups of conventional machines. To use the instruction buffers effectively as a large window, the D-morph must provide high-bandwidth instruction fetching, aggressive control and data speculation, and a high-bandwidth, low-latency memory system that preserves sequential memory semantics across a window of thousands of instructions.

3.1 Frame Space Management

By treating the instruction buffers at each ALU as a distributed issue window, orders-of-magnitude increases in window sizes are possible. This window is fundamentally a three-dimensional scheduling region, where the x- and y-dimensions correspond to the physical dimensions of the ALU array and the z-dimension corresponds to multiple instruction slots at each ALU node, as shown in Figure 2c. This three-dimensional region can be viewed as a series of *frames*, as shown in Figure 3b, in which each frame con-

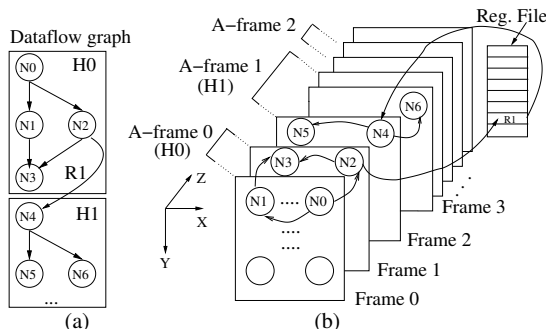


Figure 3. D-morph frame management.

sists of one instruction buffer entry per ALU node, resulting in a 2-D slice of the 3-D scheduling region.

To fill one of these scheduling regions, the compiler schedules hyperblocks into a 3-D region, assigning each instruction to one node in the 3-D space. Hyperblocks are predicated, single entry, multiple exit regions formed by the compiler [14]. A 3-D region (the array and the set of frames) into which one hyperblock is mapped is called an architectural frame, or *A-frame*.

Figure 3a shows a four-instruction hyperblock (H0) mapped into A-frame 0 as shown in Figure 3b, where N0 and N2 are mapped to different buffer slots (frames) on the same physical ALU node. All communication within the block is determined by the compiler which schedules operand routing directly from ALU to ALU. Consumers are encoded in the producer instructions as X, Y, and Z-relative offsets, as described in prior work [16]. Instructions can direct a produced value to any element within the same A-frame, using the lightweight routed network in the ALU array. The maximum number of frames that can be occupied by one program block (the maximum A-frame size) is architecturally limited by the number of instruction bits to specify destinations, and physically limited by the total number of frames available in a given implementation. The current TRIPS ISA limits the number of instructions in a hyperblock to 128, and the current implementation limits the maximum number of frames per A-frame to 16, the maximum number of A-frames to 32, and provides 128 frames total.

3.2 Multiblock Speculation

The TRIPS instruction window size is much larger than the average hyperblock size that can be constructed. The hardware fills empty A-frames with speculatively mapped hyperblocks, predicting which hyperblock will be executed next, mapping it to an empty A-frame, and so on. The A-frames are treated as a circular buffer in which the oldest A-frame is non-speculative and all other A-frames are speculative (analogous to tasks in a Multiscalar processor [19]). When the A-frame holding the oldest hyperblock completes, the block is committed and removed. The next oldest hyperblock becomes non-speculative, and the released frames can be filled with a new speculative hyperblock. On a misprediction, all blocks past the offending prediction are squashed and restarted.

Since A-frame IDs are assigned dynamically and all intra-hyperblock communication occurs within a single A-frame, each producer instruction prepends its A-frame ID to the Z-coordinate of its consumer to form the correct instruction buffer address of the consumer. Values passed between hyperblocks are transmitted through the register file, as shown by the communication of R1 from H0 to H1 in Figure 3b. Such values are aggressively forwarded when they are produced, using the register stitch table that dynamically matches the register outputs of earlier hyperblocks to the register inputs of later hyperblocks.

3.3 High-Bandwidth Instruction Fetching

To fill the large distributed window the D-morph requires high-bandwidth instruction fetch. The control model uses a program counter that points to hyperblock headers. When there is sufficient frame space to map a hyperblock, the control logic accesses a partitioned instruction cache by broadcasting the index of the hyperblock to all banks. Each bank then fetches a row's worth of instructions with a single access and streams it to the bank's respective row. Hyperblocks are encoded as VLIW-like blocks, along with a prepended header that contains the number of frames consumed by the block.

The next-hyperblock prediction is made using a highly tuned tournament exit predictor [10], which predicts a binary value that indicates the branch predicted to be the first to exit the hyperblock. The per-block accuracy of the exit predictor is shown in row 3 of Table 1; the predictor itself is described in more detail elsewhere [17]. The value generated by the exit predictor is used both to index into a BTB to obtain the next predicted hyperblock address, and also to avoid forwarding register outputs produced past the predicted branch to subsequent blocks.

3.4 Memory Interface

To support high ILP, the D-morph memory system must provide a high-bandwidth, low-latency data cache, and must maintain sequential memory semantics. As shown in Figure 2b, the right side of each TRIPS core contains distributed primary memory system banks, that are tightly coupled to the processing logic for low latency. The banks are interleaved using the low-order bits of the cache index, and can process multiple non-conflicting accesses simulta-

Benchmark	adpcm	ammp	art	bzip2	compress	dct	equake	gzip	hydro2d	m88k
Good insts/block	30.7	119	80.4	55.8	21.6	163	33.5	36.2	200	40.2
Exit/target pred. acc.	0.72	0.94	0.99	0.74	0.84	0.99	0.97	0.84	0.97	0.95
Avg. frames	2.4	5.2	3.2	2.8	1.3	6.0	2.1	3.1	7.4	2.3
# in window	116	1126	1706	364	129	1738	622	671	1573	796
Benchmark	mcf	mgrid	mpeg2	parser	swim	tomcatv	turb3d	twolf	vortex	mean
Good insts/block	29.8	179	81.3	14.6	361	210	160	48.9	29.4	99.8
Exit/target pred. acc.	0.91	0.99	0.88	0.93	0.99	0.98	0.94	0.76	0.99	0.91
Avg. frames	2.2	6.9	3.8	1.3	11.8	7.4	6.4	2.6	2.0	4.2
# in window	462	1590	958	255	1928	1629	1399	361	918	965

Table 1. Execution characteristics of D-morph codes.

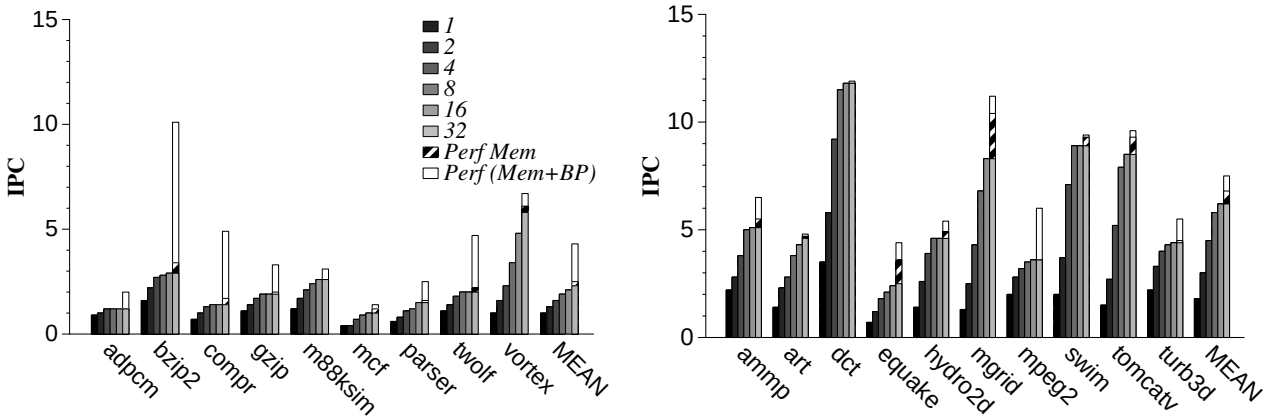


Figure 4. D-morph performance as a function of A-frame count.

neously. Each bank is coupled with MSHRs for the cache bank and a partition of the address-interleaved load/store queues that enforce ordering of loads and stores. The MSHRs, the load/store queues, and the cache banks all use the same interleaving scheme. Stores are written back to the cache from the LSQs upon block commit.

The secondary memory system in the D-morph configures the networked banks as a non-uniform cache access (NUCA) array [12], in which elements of a set are spread across multiple secondary banks, and are capable of migrating data on the two-dimensional switched network that connects the secondary banks. This network also provides a high-bandwidth link to each L1 bank for parallel L1 miss processing and fills. To summarize, with accurate exit prediction, high-bandwidth I-fetching, partitioned data caches, and concurrent execution of hyperblocks with inter-block value forwarding, the D-morph is able to use the instruction buffers as a polymorphous out-of-order issue window effectively, as shown in the next subsection.

3.5 D-morph Results

In this subsection, we measure the ILP achieved using the mechanisms described above. The results shown in this section assume a 4x4 (16-wide issue) core, with 128 physical frames, a 64KB L1 data cache that requires three cycles to access, a 64KB L1 instruction cache (both partitioned into 4 banks), 0.5 cycles per hop in the ALU array, a 10-cycle branch misprediction penalty, a 250Kb exit predictor, a 12-cycle access penalty to a 2MB L2 cache, and a 132-cycle main memory access penalty. Optimistic assumptions in the simulator currently include no modeling of TLBs or page faults, oracular load/store ordering, simulation of a centralized register file, and no issue of wrong-path instructions to the memory system. All of the binaries were compiled with the Trimaran tool set [24] (based on the Illinois Impact compiler [5]), and scheduled for the TRIPS processor with our custom scheduler/rewriter.

The first row of Table 1 shows the average number of useful dynamically executed instructions per block, discounting overhead instructions, instructions with false predicates or instructions past a block exit. The second row shows the average dynamic number of frames allocated per block by our scheduler for a 4x4 grid. Using the steady-state block (exit) prediction accuracies shown in the third row, each benchmarks holds 965 useful instructions in the distributed window, on average, as shown in row 4 of Table 1.

Figure 4 shows how IPC scales as the number of A-frames is increased from 1 to 32, permitting deeper speculative execution. The integer benchmarks are shown on the left; the floating point and Mediabench [13] benchmarks are shown on the right. Each 32 A-frame bar also has two additional IPC values, showing the performance with perfect memory in the hashed fraction of each bar, and then adding perfect branch prediction, shown in

white. Increasing the number of A-frames provides a consistent performance boost across many of the benchmarks, since it permits greater exploitation of ILP by providing a larger window of instructions. Some benchmarks show no performance improvements beyond 16 A-frames (*bzip2*, *m88ksim*, and *tomcatv*), and a few reach their peak at 8 A-frames (*adpcm*, *gzip*, *twolf*, and *hydro2d*). In such cases, the large frame space is underutilized when running a single thread, due to either low hyperblock predictability in some cases or a lack of program ILP in others.

The graphs demonstrate that while control mispredictions cause large performance losses for the integer codes (close to 50% on average), the large window is able to tolerate memory latencies extremely well, resulting in negligible slowdowns due to an imperfect memory system for all benchmarks but *mgrid*.

4 T-morph: Thread-Level Parallelism

The T-morph is intended to provide higher processor utilization by mapping multiple threads of control onto a single TRIPS core. While similar to simultaneous multithreading [23] in that the execution resources (ALUs) and memory banks are shared, the T-morph statically partitions the reservation station (issue window) and eliminates some replicated SMT structures, such as the reorder buffer.

4.1 T-Morph Implementation

There are multiple strategies for partitioning a TRIPS core to support multiple threads, two of which are *row processors* and *frame processors*. Row processors space-share the ALU array, allocating one or more rows per thread. The advantage to this approach is that each thread has I-cache and D-cache bandwidth and capacity proportional to the number of rows assigned to it. The disadvantage is that the distance to the register file is non-uniform, penalizing the threads mapped to the bottom rows. Frame processors, evaluated in this section, time-share the processor by allocating threads to unique sets of physical frames. We describe the polymorphous capabilities required for each of the classes of mechanisms below.

Frame space management: Instead of holding non-speculative and speculative hyperblocks for a single thread as in the D-morph, the physical frames are partitioned *a priori* and assigned to threads. For example, a TRIPS core can dedicate all 128 frames to a single thread in the D-morph, or 64 frames to each of two threads in the T-morph (uneven frame sharing is also possible). Within each thread, the frames are further divided into some number of A-frames and speculative execution is allowed within each thread. No additional register file space is required, since the same storage used to hold state for speculative blocks can instead store state from multiple non-speculative and speculative blocks. The only additional frame support needed is thread-ID bits in the register stitching logic and augmentations to the A-frame allocation logic.

Instruction control: The T-morph maintains n program counters (where n is the number of concurrent threads allowed) and n global history shift registers in the exit predictor to reduce thread-induced mispredictions. The T-morph fetches the next block for a given thread using a prediction made by the shared exit predictor, and maps it onto the array. In addition to the extra prediction registers, n copies of the commit buffers and block control state must be provided for n hardware threads.

Memory: The memory system operates much the same as the D-morph, except that per-thread IDs on cache tags and LSQ CAMs are necessary to prevent illegal cross-thread interference, provided that shared address spaces are implemented.

4.2 T-morph Results

To evaluate the performance of multi-programmed workloads running on the T-morph, we classified the applications as "high memory intensive" and "low memory intensive", based on L2 cache miss rates. We picked eight different benchmarks and ran different combinations of 2, 4 and 8 benchmarks executing concurrently. The high memory intensive benchmarks are art_h, mcf_h, $equake_h$, and $tomcatv_h$. The low memory intensive benchmarks are $compress_l$, $bzip2_l$, $parser_l$, and $m88ksim_l$. We examine the performance obtained while executing multiple threads concurrently and quantify the sources of performance degradation. Compared to a single thread executing in the D-morph, running threads concurrently introduces the following sources of performance loss: a) inter-thread contention for ALUs and routers in the grid, b) cache pollution, c) pollution and interaction in the branch predictor tables, and d) reduced speculation depth for each thread, since the number of available frames for each thread is reduced.

Table 2 shows T-morph performance on a 4x4 TRIPS core with parameters similar to those of the baseline D-morph. The second column lists the combined instruction throughput of the running threads. The third column shows the sum of the IPCs of the benchmarks when each is run on a separate core but with same number of frames as available to each thread in the T-morph. Comparing the throughput of column 3 with the throughput in column 2, indicates the performance drop due to inter-thread interaction in the T-morph. Column 4 shows the cumulative IPCs of the threads when each is run by itself on a TRIPS core with all frames available to it. Comparison of this column with column 4, indicates the performance drop incurred from both inter-thread interaction and reduced speculation

Benchmarks	Throughput (aggregate IPC)			Overall Efficiency (%)	Per Thread Efficiency (%)	Speedup
	T-morph	Constant A-frames	Scaled A-frames			
2 Threads						
$bzip_l, m88ksim_l$	4.9	5.5	5.5	90	93, 86	1.8
$parser_l, m88ksim_l$	3.7	3.8	4.1	90	88, 91	1.8
$art_h, compress_l$	5.1	5.7	6.0	86	93, 62	1.6
$mcf_h, bzip_l$	3.2	3.9	3.9	81	98, 75	1.7
art_h, mcf_h	5.1	5.3	5.6	90	91, 87	1.8
$equake_h, mcf_h$	3.3	3.4	3.5	95	101, 83	1.8
MEAN	4.7	-	-	87	84	1.7
4 Threads						
$bzip_l, m88ksim_l, parser_l, compress_l$	6.1	6.7	8.4	72	79, 70, 59, 78	2.9
$equake_h, art_h, parser_l, compress_l$	6.1	7.0	10.0	61	68, 69, 38, 47	2.2
$tomcatv_h, mcf_h, m88ksim_l, bzip_l$	8.3	10.7	15.0	55	54, 65, 55, 58	2.3
$equake_h, art_h, tomcatv_h, mcf_h$	9.0	10.5	16.6	54	60, 58, 51, 53	2.2
MEAN	7.4	-	-	61	60	2.4
8 Threads						
$art, tomcatv, bzip, m88ksim$ $equake, parser, compress, mcf$	9.8	17.7	25.0	39	40, 44, 34, 33 50, 23, 26, 43	2.9

Table 2. T-morph thread efficiency and throughput.

in the T-morph. Our experiments showed that T-morph performance is largely insensitive to cache and branch predictor pollution, but is highly sensitive to instruction fetch bandwidth stalls.

Column 5 shows the overall T-morph efficiency, defined as the ratio of multithreading performance to throughput of threads running on independent cores (column 2/column 4). Column 6 breaks this down further showing the fraction of peak D-morph performance achieved by each thread when sharing a TRIPS core with other threads. The last column shows an estimate of the speedup provided by the T-morph versus running each of the applications one at a time on a single TRIPS core (with the assumption that each application has approximately the same running time). The overall efficiency varies from 80–100% with 2 threads down to 39% with 8 threads. Having the low memory benchmarks resident simultaneously provided the highest efficiency, while mixes of high memory benchmarks provided the lowest efficiency, due to increased T-morph cache contention. This effect is less pronounced in the 2-thread configurations with the pairing of high memory benchmarks being equally efficient as others. The overall speedup provided by multithreading ranges from a factor 1.4 to 2.9 depending on the number of threads. In summary, most benchmarks do not completely exploit the deep speculation provided by all of the A-frames available in the D-morph, due to branch mispredictions. The T-morph converts these less useful A-frames to non-speculative computations when multiple threads or jobs are available. Future work will evaluate the T-morph on multithreaded parallel programs.

5 S-morph: Data-Level Parallelism

The S-morph is a configuration of the TRIPS processor that leverages the technology scalable array of ALUs and the fast inter-ALU communication network for streaming media and scientific applications. These applications are typically characterized by data-level parallelism (DLP) including predictable loop-based control flow with large iteration counts [21], large data sets, regular access patterns, poor locality but tolerance to memory latency, and high computation intensity with tens to hundreds of arithmetic operations performed per element loaded from memory [18]. The S-morph was heavily influenced by the Imagine architecture [11] and uses the Imagine execution model in which a set of stream kernels are sequenced by a control thread. Figure 5 highlights the features of the S-morph which are further described below.

5.1 S-morph Mechanisms

Frame Space Management: Since the control flow of the programs is highly predictable, the S-morph fuses multiple A-frames to make a *super A-frame*, instead of using separate A-frames for speculation or multithreading. Inner loops of a streaming application are unrolled to fill the reservation stations within these super A-frames. Code required to set up the execution of the inner loops and to connect multiple loops can run in one of three ways: (1) embedded into the program that uses the frames for S-morph execution, (2) executed on a different core within the TRIPS chip–similar in function to the Imagine host processor, or (3) run within its own set of frames on the same core running the DLP kernels. In this third mode, a subset of the frames are dedicated to a data parallel thread, while a different subset are dedicated to a sequential control thread.

Instruction Fetch: To reduce the power and instruction fetch bandwidth overhead of repeated fetching of the same code block across inner-loop iterations, the S-morph employs *mapping reuse*, in which a block is kept in the reservation stations and used multiple times. The S-morph im-

Benchmark	Original Iteration			Fused Iterations				# of Revitalizations
	Kernel size (insts)	Inputs/ Outputs	Constants	Unrolling factor	Compute insts per block	Block size	Total Constants	
convert	15	3/3	9	16	240	303	144	171
dct	70	8/8	10	8	560	580	80	128
fir16	34	1/1	16	16	544	620	256	512
fft8	104	16/16	16	4	416	570	64	128
idea	112	2/2	52	8	896	1020	416	512
transform	37	8/8	21	16	592	740	336	64

Table 3. Characteristics of S-morph codes.

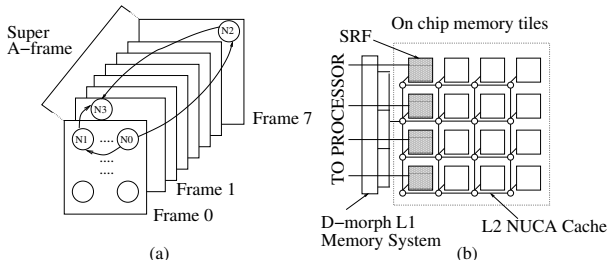


Figure 5. Polymorphism for S-morph.

plements mapping reuse with a <repeat N> instruction (similar to RPTB in the TMS320C54x [1]) which indicates that the next block of instructions constitute a loop and is to execute a finite number of times N where N can be determined at runtime and is used to set an iteration counter. When all of the instructions from an iteration complete, the hardware decrements the iteration counter and triggers a *revitalization signal* which resets the reservation stations, maintaining constant values residing in reservation station, so that they may fire again when new operands arrive for the next iteration. When the iteration counter reaches zero, the super A-frame is cleared and the hardware maps the next block onto the ALUs for execution.

Memory System: Similar to Smart Memories [15], the TRIPS S-morph implements the Imagine stream register file (SRF) using a subset of on-chip memory tiles. S-morph memory tile configuration includes turning off tag checks to allow direct data array access and augmenting the cache line replacement state machine to include DMA-like capabilities. Enhanced transfer mechanisms include block transfer between the tile and remote storage (main memory or other tiles), strided access to remote storage (gather/scatter), and indirect gather/scatter in which the remote addresses to access are contained within a subset of the tile's storage. Like the Imagine programming model, we expect that transfers between the tile and remote memory will be orchestrated by a separate thread.

As shown in Figure 5b, memory tiles adjacent to the processor core are used for the SRF and are augmented with dedicated wide channels (256 bits per row assuming 4 64-bit channels for the 4x4 array) into the ALU array for increased SRF bandwidth. The S-morph DLP loops can execute an SRF_read that acts as *load multiple word* instruction by transferring an entire SRF line into the grid, spreading it across the ALUs in a fixed pattern within a row. Once within the grid, data can be easily moved to any ALU using the high-bandwidth in-grid routing network, rather than requiring a data switch between the SRF banks and the ALU array. Streams are striped across the multiple banks of the SRF. Stores to the SRF are aggregated in a store buffer and then transmitted to the SRF bank over narrow channels to the memory tile. Memory tiles not adjacent to the processing core can be configured as a conventional level-2 cache still accessible to the unchanged level-1 cache hierarchy. The conventional cache hierarchy can be used to store irregularly accessed data structures, such as texture maps.

5.2 Results

We evaluate the performance of the TRIPS S-morph on a set of streaming kernels, shown in Table 3, extracted from the Mediabench benchmark suite [13]. These kernels were selected to represent different computation-to-memory ratios, varying from less than 1 to more than 14. The kernels are hand-coded in a TRIPS meta-assembly language, then mapped to the ALU array using a custom scheduler akin to the D-morph scheduler, and simulated using an event-driven simulator that models the TRIPS S-morph.

Program characteristics: Columns 2–4 of Table 3 show the intrinsic characteristics for one iteration of the kernel code, including the number of arithmetic operations, the number of bytes read from/written to memory, and number of unique run time constants required. The unrolling factor for each inner loop is determined by the size of the kernel and the capacity of the super A-frame (a 4x4 grid with 128 frames or 2K instructions). The useful instructions per block includes only computation instructions while the block size numbers include overhead instructions for memory access and data movement within the grid. The total constant count indicates the number of reservation stations that must be filled with constant values from the register file for each iteration of the unrolled loop. Most

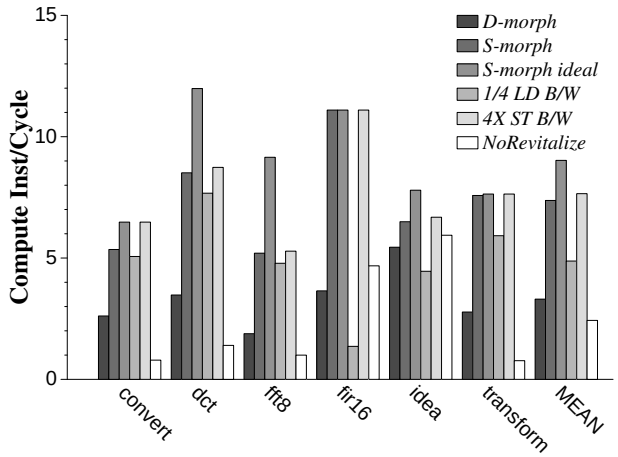


Figure 6. S-morph performance.

of these register moves can be eliminated by allowing the constants to remain in reservation stations across revitalizations. The number of revitalizations corresponds to the number of iterations of the unrolled loop. The unrolling of the kernels is based on 64-Kbyte input and output streams, both being striped and stored in the SRF.

Performance analysis: Figure 6 compares the performance of the D-morph to the S-morph on a 4x4 TRIPS core with 128 frames, a 32 entry store buffer, 8 cycle revitalization delay, and a pipelined 7-cycle SRF access delay. The D-morph configuration in this experiment assumes perfect L1 caches with 3-cycle hit latencies. Figure 6 shows that the S-morph sustains an average of 7.4 computation instructions per cycle (not counting overhead instructions or address compute instructions), a factor of 2.4 higher than the D-morph. A more idealized S-morph configuration that employs 256 frames and no revitalization latency improves performance to 9 compute ops/cycle, 26% higher than the realistic S-morph. An alternative approach to S-morph polymorphism is the Tarantula architecture [8] which exploits data-level parallelism by augmenting the processor core of an Alpha 21464 with a dedicated vector data path of 32 ALUs, an approach that sustains between 10 and 20 FLOPS per cycle. Our results indicate that the TRIPS S-morph can provide competitive performance on data-parallel workloads; a 8x4 grid consisting of 32 ALUs sustains, on average, 15 compute ops per cycle. Furthermore, the polymorphous approach provides superior area efficiency compared to Tarantula, which contains two large heterogeneous cores.

SRF bandwidth: To investigate the sensitivity of the S-morph to SRF bandwidth we investigated two alternative design points: load bandwidth decreased to 64 bits per row (*1/4 LD B/W*) and store bandwidth increased to 256 bits per row (*4X ST B/W*). Decreasing the load bandwidth drops performance by 5% to 31%, with a mean percentage drop of 27%. Augmenting the store bandwidth increases average IPC to 7.65 corresponding to 5% performance improvement on average. However, on a 8x8 TRIPS core, experiments show that increased store bandwidth can improve performance by 22%. As expected, compute intensive kernels, such as *fir* and *idea*, show little sensitivity to SRF bandwidth.

Revitalization: As shown in the *NoRevitalize* bar in Figure 6, eliminating revitalization causes S-morph performance to drop by a factor of 5 on average. This effect is due to the additional latency for mapping instructions into the grid as well as redistributing the constants from the register file on every unrolled iteration. For example, the unrolled inner loop of the *dct* kernel requires 37 cycles to fetch the 580 instructions (assuming 16 instructions fetched per cycle) plus another 10 cycles to fetch the 80 constants from the banked register file. Much of this overhead is exposed because unlike the D-morph with its speculative instruction fetch, the S-morph has hard synchronization boundaries between iterations. One solution that we are examining to further reduce the impact of instruction fetch is to overlap revitalization and execution. Further extensions to this configuration can allow the individual ALUs at each node to act as separate MIMD processors. This technique would benefit applications with frequent data-dependent control flow, such as real-time graphics and network processing workloads.

6 Scalability to Larger Cores

While the experiments in Sections 3 – 5 reflect the performance achievable on three application classes with 16 ALUs, the question of granularity still remains. Given a fixed single-chip silicon budget, how many processors should be on the chip, and how powerful should each processor be? To address this question, we first examined the performance of each application class as a function of the architecture granularity by varying the issue width of a TRIPS core. We use this information to determine the sweet spot for each application class and then describe how this sweet spot can be achieved for each application class using the configurability of the TRIPS system.

Figures 7a and 7b show the aggregate performance of ILP and DLP workloads on TRIPS cores of different dimensions, including 2x2, 4x4, 8x4, and 8x8. The selected benchmarks represent the general behavior of the benchmark suite as a whole. Unsurprisingly, the benchmarks with low instruction-level concurrency see little benefit from TRIPS cores larger than 4x4, and a class of them (represented by *adpcm*) sees little benefit beyond 2x2. Benchmarks with higher concurrency such as *swim* and *idea* see diminishing returns beyond 8x4, while others, such as *mgrid* and *fft* continue to benefit from increasing

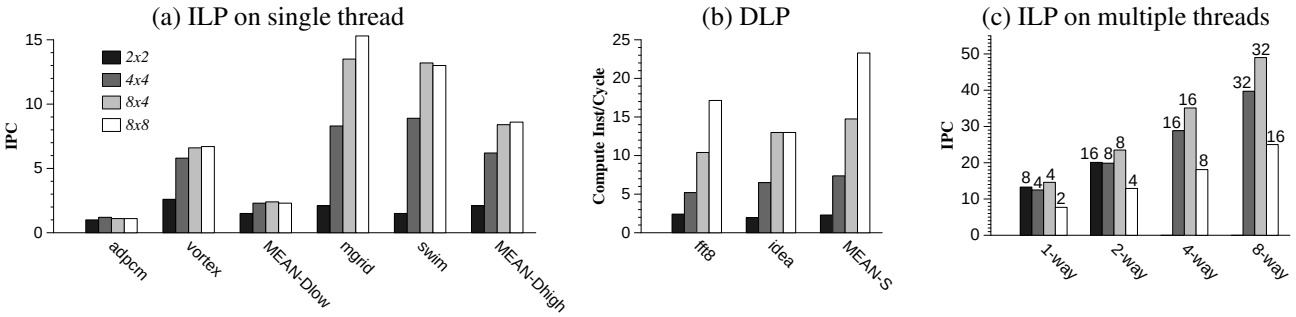


Figure 7. TRIPS single-core scalability and CMP throughput.

ALU density. Table 4 shows the best suited configurations for the different applications in column 2.

The variations across applications and application domains demand both large coarse-grain processors (8x4 and 8x8) and small fine-grain processors (2x2). Nonetheless, for single-threaded ILP and DLP applications, the larger processors provide better aggregate performance at the expense of low utilization for some applications. For multithreaded and multiprogrammed workloads, the decision is more complex. Table 4 shows several alternative TRIPS chip designs, ranging from 8 2x2 TRIPS cores to 2 8x8 cores, assuming a $400mm^2$ die in a 100nm technology. The equivalent real estate could be used to construct 10 Alpha 21264 processors and 4MB of on-chip L2 cache.

Figure 7c shows the instruction throughput (in aggregate IPC), with each bar representing the core dimensions, each cluster of bars showing the number of threads per core, and the number atop each bar showing the total number of threads (# cores times threads per core). The 2x2 array is the worst performing when large number of threads are available. The 4x4 and 8x4 configurations have the same number of cores due to changing on-chip cache capacity, but the 8x4 and 8x8 have the same total number of ALUs and instruction buffers across the full chip. With ample threads and at most 8 threads per core, the best design point is the 8x4 topology, no matter how many total threads are available (e.g., of all the bars labeled 16 threads, the 8x4 configuration is the highest-performing). These results validate the large-core approach; one 8x4 core has higher performance for both single-threaded ILP and DLP workloads than a smaller core, and shows higher throughput than many smaller cores using the same area when many threads are available. We are currently exploring and evaluating space-based subdivision for both TLP and DLP applications beyond the time-based multithreading approach described in this paper.

Grid Dimensions	Preferred Applications	# TRIPS cores	Total L2 (MB)
2x2	adpcm	8	3.90
4x4	vortex	4	3.97
8x4	swim, idea	4	1.25
8x8	mgrid, fft	2	1.25
21264	-	10	3.97

Table 4. TRIPS CMP Designs.

7 Conclusions and Future Directions

The polymorphous TRIPS system enables a single set of processing and storage elements to be configured for multiple application domains. Unlike prior configurable systems that aggregate small primitive components into larger processors, TRIPS starts with a large, technology-scalable core that can be logically subdivided to support ILP, TLP, and DLP. The goal of this system is to achieve performance and efficiency approaching that of special-purpose systems. In this paper, we proposed a small set of mechanisms (managing reservation stations and memory tiles) for a large-core processor that enables adaptation into three modes for these diverse application domains. We have shown that all three modes achieve the goal of high performance on their respective application domain. The D-morph sustains 1–12 IPC (average of 4.4) on serial codes, the T-morph achieves average thread efficiencies of 87%, 60%, and 39% for two, four, and eight threads, respectively, and the S-morph executes as many as 12 arithmetic instructions per clock on a 16-ALU core, and an average of 23 on an 8x8 core.

While we have described the TRIPS system as having three distinct personalities (the D, T, and S-morphs), in reality each of these configurations is composed of basic mechanisms that can be mixed and matched across execution models. In addition, there are also minor reconfigurations, such as adjusting the level-2 cache capacity, that do not require a change in the programming model. A major challenge for polymorphous systems is designing the interfaces between the software and the configurable hardware as well as determining when and how to initiate reconfiguration. At one extreme, application programmers and compiler writers can be given a fixed number of static morphs; programs are written and compiled to these static machine models. At the other extreme, a polymorphous

system could expose all of the configurable mechanisms to the application layers, enabling them to select the configurations and the time of reconfiguration. We are exploring both the hardware and software design issues in the course of our development of the TRIPS prototype system.

Acknowledgments

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF instrumentation grant EIA-9985991, NSF CAREER grants CCR-9985109 and CCR-9984336, two IBM University Partnership awards, and grants from the Alfred P. Sloan Foundation, the Peter O'Donnell Foundation, and the Intel Research Council.

References

[1] TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set, Literature Number: SPRU172C, March 2001.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[3] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – A Self-Reconfigurable Data Processing Architecture. In *1st International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2001.

[4] C. Casçaval, J. Castanos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. W. Jr. Evaluation of multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 311–322, January 2002.

[5] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, May 1991.

[6] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[7] C. Ebeling, D. C. Cronquist, and P. Franklin. Configurable computing: The catalyst for high-performance architectures. In *International Conference on Application-Specific Systems, Architectures, and Processors*, pages 364–372, 1997.

[8] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of The 29th International Symposium on Computer Architecture*, pages 281–292, May 2002.

[9] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[10] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, Feb. 1997.

[11] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

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

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[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 *Proceedings of the 25st International Symposium on Microarchitecture*, pages 45–54, 1992.

[15] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[16] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 40–51, December 2001.

[17] N. Ranganathan, R. Nagarajan, D. Burger, and S. W. Keckler. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, September 2002.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings on the 31st International Symposium on Microarchitecture*, pages 3–13, December 1998.

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

[20] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[21] D. Talla, L. John, and D. Burger. Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements. *IEEE Transactions on Computers, to appear*, pages 35–46, 2003.

[22] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–26, January 2001.

[23] 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*, pages 392–403, June 1995.

[24] V.Kathail, M.Schlansker, and B.R.Rau. Hpl-pd architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

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