

# TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP

KARTHIKEYAN SANKARALINGAM

RAMADASS NAGARAJAN

HAIMING LIU

CHANGKYU KIM

JAEHYUK HUH

NITYA RANGANATHAN

DOUG BURGER

STEPHEN W. KECKLER

ROBERT G. MCDONALD

CHARLES R. MOORE

The University of Texas at Austin

---

This article describes the **polymorphous** TRIPS architecture which can be configured for different granularities and types of parallelism. The TRIPS architecture is the first in a class of post-RISC, dataflow-like instruction sets called Explicit Data-Graph Execution (EDGE). This EDGE ISA is coupled with hardware 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 prototype contains two 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.

Categories and Subject Descriptors: C.1 [Processor Architectures]: ; C.1.1 [Single Data Stream Architectures]: ; C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]: ; C.1.3 [Other Architecture Styles]:

General Terms: Computer systems

Additional Key Words and Phrases: Computer architecture, configurable computing, scalable and high-performance computing

---

Authors' address: Department of Computer Sciences, 1 University Station C0500, The University of Texas at Austin, Austin, TX 78712-1188 (*cart@cs.utexas.edu*)

This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF instrumentation grant EIA-9985991, NSF CAREER awards CCR-9985109 and CCR-9984336, two IBM University Partnership awards, and grants from the Alfred P. Sloan Foundation, the O'Donnell Foundation, Sun Microsystems, and the Intel Research Council. Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 \$5.00

## 1. INTRODUCTION

General-purpose microprocessors owe their success to their ability to run many diverse workloads well. However, current-generation general-purpose ISAs do not lend themselves to the rich diversity of current and emerging applications. Today, many application-specific processors, such as network, server, scientific, graphics, and digital signal processors have been constructed to match the particular parallelism characteristics of their application domains. A truly “application general-purpose” processor—that could run not only single-threaded programs but also exploit many types of concurrency—would have substantive benefits, including improved system flexibility as well as reduced design and mask costs across greater economies of scale.

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) based on conventional ISAs, 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 with specialized architectures, each designed to run a distinct class of workloads effectively. The proposed Tarantula processor is one such example of integrated heterogeneity [Espasa et al. 2002]. 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 contain an architecture, including an instruction set, that can 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.

Two key questions are (1) what granularity of processors and memories on a CMP is best for polymorphous capabilities, and (2) for a given granularity, what processor architecture would best support high flexibility. 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  $400\text{mm}^2$  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:

- a) Ultra-fine-grained FPGAs consisting of programmable interconnect and lookup

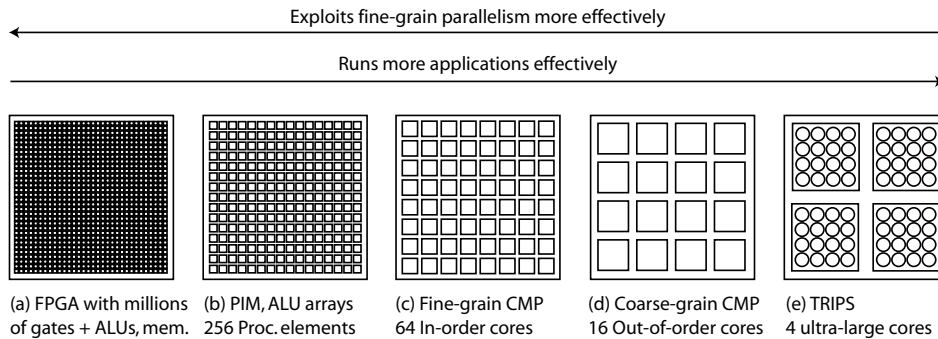


Fig. 1. Granularity of parallel processing elements on a chip.

- tables, as well as increasing availability of memory and ALU macros [Xilinx].
- b) Hundreds of primitive processors connected to memory banks such as a processor-in-memory (PIM) architecture [Kang et al. 1999; Oskin et al. 1998; Sterling and Zima 2002] or reconfigurable ALU arrays such as RaPiD [Ebeling et al. 1997], Piperench [Goldstein et al. 2000], and PACT [Baumgarte et al. 2001]. While the topologies of these architectures are not always regular arrays, the basic organizations can still be represented by (b).
  - c) Tens of simple in-order processors, such as in RAW [Waingold et al. 1997] or Piranha [Barroso et al. 2000] architectures.
  - d) Coarse grained architectures consisting of 10-20 4-issue cores, such as the Power4 [Tendler et al. 2001], Cyclops [Casçaval et al. 2002], MultiScalar processors [Sohi et al. 1995], other proposed speculatively-threaded CMPs [Cintra et al. 2000; Steffan et al. 2000], and the polymorphous Smart Memories [Mai et al. 2000] architecture.
  - e) Wide-issue processors with many ALUs each, such as Grid Processors [Nagarajan et al. 2001], a specific instance of which is the TRIPS Processor.

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 theoretical peak performance, but its performance on control-bound codes with irregular memory accesses, such as sparse matrix scientific codes or program compilation, would be dismal at best. At the other extreme, coarser-grained architectures traditionally have not had the capability to use their internal computational resources to show high performance on fine-grained, highly parallel applications.

Polymorphism can bridge this dichotomy with either of two competing approaches. A *synthesizing* 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 provides the configuration and ISA support to partition the large processors logically, exploiting finer-grain parallelism when it is present.

Regardless of the approach, a polymorphous architecture will not outperform custom hardware dedicated to a specific 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 a customized solution for each application.

This paper proposes and describes the polymorphous TRIPS architecture, which uses the partitioning approach, combining an instance of coarse-grained, polymorphous Grid Processor cores with an adaptive, polymorphous on-chip memory system. One goal is to design cores that are both as large as desirable, 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. Conversely, the competing approach of synthesizing coarser-grain processors from fine-grained components must overcome the significant challenges of distributed control, long interaction latencies, and synchronization overheads, which have all proven challenging to date.

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 the planned TRIPS silicon prototype, its instruction set architecture, 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 for desktop applications with the D-morph (Section 3), thread-level parallelism with the threaded T-morph (Section 4), and data-level parallelism with the streaming S-morph (Section 5). Section 6 quantifies 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 8 that by building large, partitionable, 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 employs large, coarse-grained grid processors (GPA) to achieve high performance on single-threaded applications with high ILP. The processor cores employ an explicit data-graph execution (EDGE) instruction set, as we describe below. These cores are augmented with polymorphous features that enable the compiler or run-time system to subdivide the core 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 these 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 TRIPS processor cores and memory system are essentially pools of distributed ALUs and memory banks, which contain the architectural support

necessary for software to place varied mappings and flows of computation and data. To clarify the terminology we use with an analogy, the prototype *TRIPS processor* is an implementation of the *TRIPS architecture*, just as a processor like the IBM PowerPC 970 is an implementation of the PowerPC architecture. The TRIPS processor is a member of the *Grid Processor* family of designs, just as the 970 is a member of the superscalar family of designs, and the TRIPS ISA is an example of an *EDGE ISA*, just as the PowerPC architecture is an example of a RISC ISA.

The key challenge in defining polymorphous features for TRIPS is to balance 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.

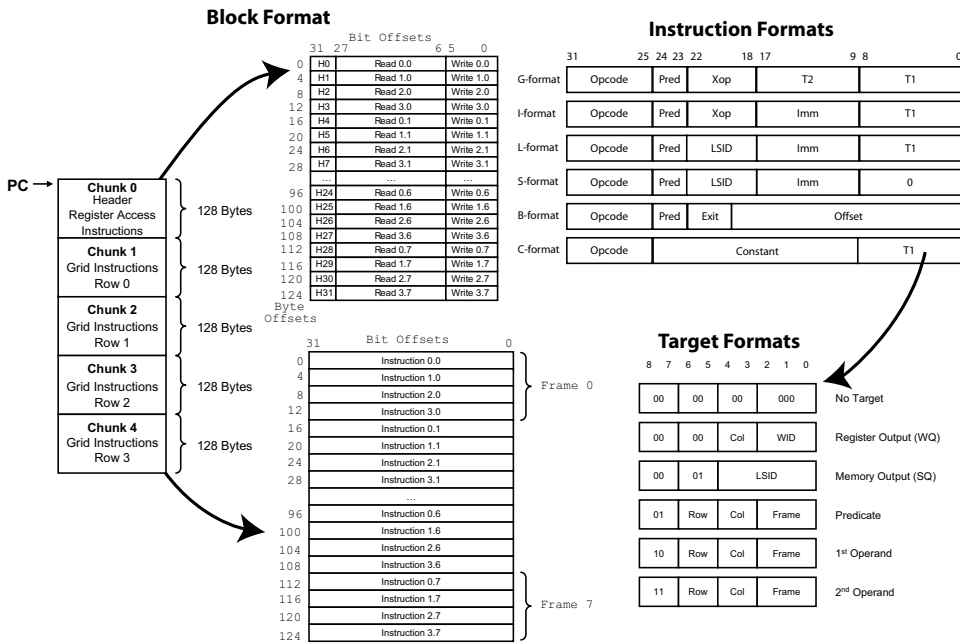
## 2.1 Core Execution Model

The TRIPS architecture is fundamentally *block oriented*, and builds on prior proposals for block-structured ISAs [Hao et al. 1996]. 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 [Mahlke et al. 1992]. For instruction and thread-level parallel programs, blocks commit atomically and interrupts are *block precise*, meaning that they are handled only at block boundaries. Each block has both a static set of state inputs and a potentially variable set of state outputs that depends upon the internal predicates and which exit is taken 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.

For all modes of execution, the compiler is responsible for statically placing each block of instructions onto the computational engine such that inter-instruction dependences are explicit. Within a block, the instruction set is a fine-grained *dataflow instruction set*, where instructions contain the links between producing and consuming pairs instructions, rather than going through an intermediate name (such as a register) as in conventional architectures. We refer to this class of architectures as *Explicit Dataflow Graph Execution* (EDGE), as opposed to more conventional architecture families such as RISC or CISC.

Figure 2 shows the current definition of the TRIPS prototype’s instruction set architecture. Each block contains a 128-byte block header, which specifies the number of block outputs, as well as the registers that are read from and written to by the block. The rest of the block consists of four 128-byte instruction “chunks,” each of which services 32 32-bit instructions to a row. Each row contains 4 nodes with 8 instructions per node.

The ISA provides the six instruction formats shown in Figure 2, which differ based on instruction function and number of distinct instruction targets. The major dif-



**Instructions**

Load and Store Instructions		
LB	Load Byte	L
LH	Load Halfword	L
LW	Load Word	L
LD	Load Doubleword	L
SB	Store Byte	S
SH	Store Halfword	S
SW	Store Word	S
SD	Store Doubleword	S
Integer Arithmetic Instructions		
ADD	Add	G
ADDI	Add Immediate	I
SUB	Subtract	G
SUBI	Subtract Immediate	I
MUL	Multiply	G
MULI	Multiply Immediate	I
DIVS	Divide Signed	G
DIVSI	Divide Signed Immediate	I
DIVU	Divide Unsigned	G
DIVUI	Divide Unsigned Immediate	I
Integer Logical Instructions		
AND	Bitwise AND	G
ANDI	Bitwise AND Immediate	I
OR	Bitwise OR	G
ORI	Bitwise OR Immediate	I
XOR	Bitwise XOR	G
XORI	Bitwise XOR Immediate	I
Integer Shift Instructions		
SLL	Shift Left Logical	G
SLLI	Shift Left Logical Immediate	I
SRL	Shift Right Logical	G
SRLI	Shift Right Logical Immediate	I
SRA	Shift Right Arithmetic	G
SRAI	Shift Right Arithmetic Immediate	I
Integer Extend Instructions		
EXTSB	Extend Signed Byte	G
EXTSH	Extend Signed Halfword	G
EXTSW	Extend Signed Word	G
EXTUB	Extend Unsigned Byte	G
EXTUH	Extend Unsigned Halfword	G
EXTUW	Extend Unsigned Word	G
Integer Test Instructions		
TEQ	Test EQ	G
TEQI	Test EQ Immediate	I
TLT	Test LT	G
TLTI	Test LT Immediate	I
TLE	Test LE	G
TLEI	Test LE Immediate	I
TLTU	Test LT Unsigned	G
TLTUI	Test LT Unsigned Immediate	I
TLEU	Test LE Unsigned	G
TLEUI	Test LE Unsigned Immediate	I
Floating-Point Arithmetic Instructions		
FADD	FP Add	G
FSUB	FP Subtract	G
FMUL	FP Multiply	G
FDIV	FP Divide	G
Floating-Point Test Instructions		
FEQ	FP Test EQ	G
FLT	FP Test LT	G
FLE	FP Test LE	G
Floating-Point Conversion Instructions		
FITOD	Convert Integer to Double FP	G
FDTOI	Convert Double FP to Integer	G
FSTOD	Convert Single FP to Double FP	G
FDTOS	Convert Double FP to Single FP	G
Control Flow Instructions		
BR	Branch	B
BRO	Branch with Offset	B
CALL	Call	B
CALLO	Call with Offset	B
RET	Return	B
SCALL	System Call	B
Miscellaneous Instructions		
NULL	Nullify Output	G
MOV	Move	G
MOVI	Move Immediate	I
GENS	Generate Signed Constant	C
GENU	Generate Unsigned Constant	C
APP	Append Constant	C
NOP	No Operations	C

Fig. 2. The TRIPS Instruction Set Architecture.

ference between the TRIPS EDGE instructions and conventional RISC instructions is that each instruction contains no source operand specifiers, but instead contains the name(s) of its consumer instructions. These consumer names are called *targets*; one or two per value-producing instruction are provided. If an instruction has more targets in a block than can be specified in the instruction format, extra instructions must be inserted to forward the value to the additional targets, resulting in a software fan-out tree. This “push” model permits the compiler to schedule dataflow-like execution within a block on the underlying substrate. Communication among different blocks must still occur through registers. As shown in Figure 2, targets in the TRIPS ISA consist of nine bits. Seven of the bits specify the target instruction number, which corresponds either to a physical reservation station in the execution array, or as a block output to one of the 128 architectural registers. The remaining two bits specify whether each target is an architectural register or is in the block, and if it is in the block, whether it is the left, right, or predicate operand of the consuming instruction.

In addition to the standard immediate and extended opcode fields, there are several other non-standard fields in the instruction set. Each instruction contains a two-bit predicate field, which specifies whether the instruction should execute as soon as its operands arrive, or whether it should wait for a predicate operand to arrive before firing. In the latter case, the instruction fires only if the arriving predicate matches its predicate type (true or false). Having binary predicate values reduces the number of overhead instructions needed to invert predicates. Finally Memory and branch instructions have special tags; loads and stores have tags (LSIDs) to specify their program order to the memory system, and branches contain tags to assist the block-level branch predictor.

## 2.2 Architectural Overview

Figure 3a shows a diagram of the TRIPS prototype chip architecture currently being designed. This chip will consist of two polymorphous, 16-wide out-of-order issue cores, an array of 32 64KB 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 130nm process and is targeted for completion in 2005. Due to the partitioned structures and short point-to-point wiring connections, the architecture is scalable to larger dimensions and higher clock rates than will be achieved in the first prototype.

Figure 3b 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 [Nagarajan et al. 2001], 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 storage for an instruction, two source operands, and a predicate field. 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.

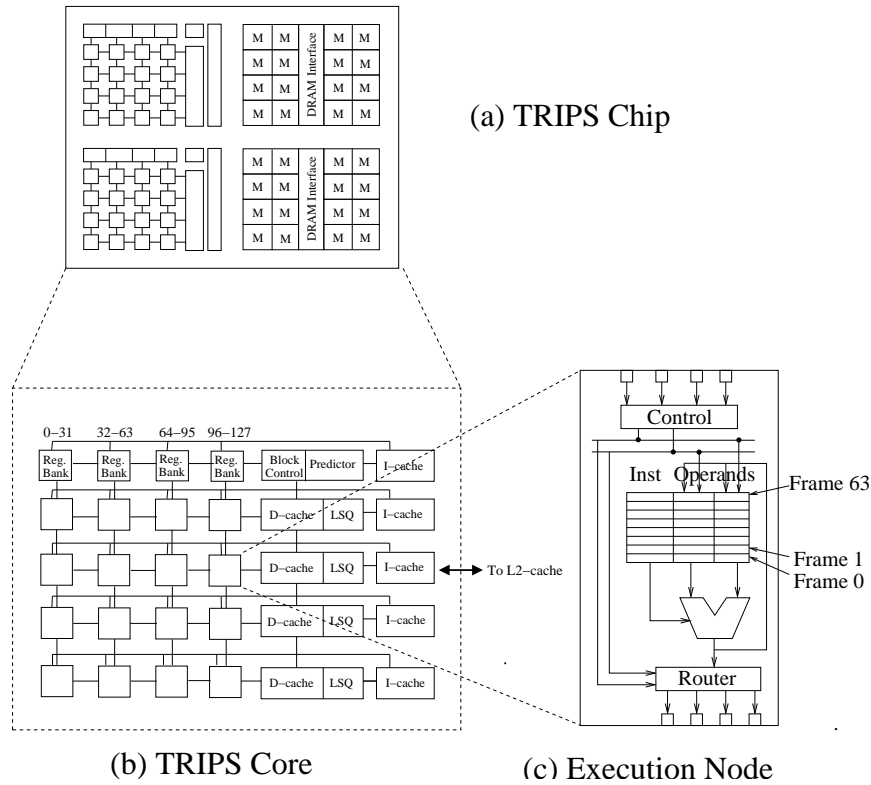


Fig. 3. TRIPS architecture overview.

The banked instruction cache on the right side couples one bank per row, with an additional instruction cache bank to hold block headers, which include the encoded instructions that move values from registers for injection into the ALU array. The banked register file above the ALU array holds a portion of the architectural state. Also to the right of the execution nodes are a set of banked, cache-line interleaved level-1 data caches, which can be accessed by any ALU through the local ALU routing network. In the upper right-hand corner of the processor is the block control logic tile that is responsible for sequencing block execution and selecting the next block. The control tile also contains the L1 instruction cache tags. The back side of the L1 caches are connected to secondary memory tiles through the two-dimensional, switched on-chip interconnection network (OCN). The OCN provides a robust and scalable connection to a large number of memory tiles, using less wiring than conventional, dedicated point-to-point channels among these components.

The TRIPS architecture contains three main types of resources. First, the hard-coded, non-polymorphous *common 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 among 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 dif-

ferently depending on the mode. The third type are *specialized resources*, which are not required for all modes and can be disabled when not in use for a given mode. Examples of specialized resources include extra program counters and branch history registers, used for additional threads in the T-morph, and high-bandwidth data channels in the S-morph. Sections 3–5 discuss these limited specialized resources further for each of the morphs.

### 2.3 Polymorphous Resources

- Frame space: As shown in Figure 3c, 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 prototype, which has a 4x4 array of ALUs with 64 reservation stations per ALU, thus has 64 frames of 16 instructions each. The *frame space*, or collection of frames, is a polymorphous resource in TRIPS, as it is managed differently by each mode 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 registers. The extra copies can be used in different ways, such as for speculation or multithreading, depending on the mode of operation. In the D-morph, these additional registers hold speculative state for the speculative instruction blocks executing concurrently with the non-speculative instruction block. In the T-morph, some of these registers hold non-speculative state for the multiple simultaneously executing threads.
- 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, or to prioritize between multiple concurrently executing threads, which is useful for multithreaded parallel programs.
- Memory tiles: The TRIPS Memory tiles can be configured to behave as NUCA style L2 cache banks [Kim et al. 2002], scratchpad memory, or 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.

Although the TRIPS instruction set architecture is quite different from conventional architectures, it supports many more types of computational patterns efficiently than does a conventional RISC ISA. In the following sections, we show how to combine this dataflow-like architecture with the TRIPS polymorphic mechanisms to exploit high levels of instruction-level, thread-level, and data-level parallelism.

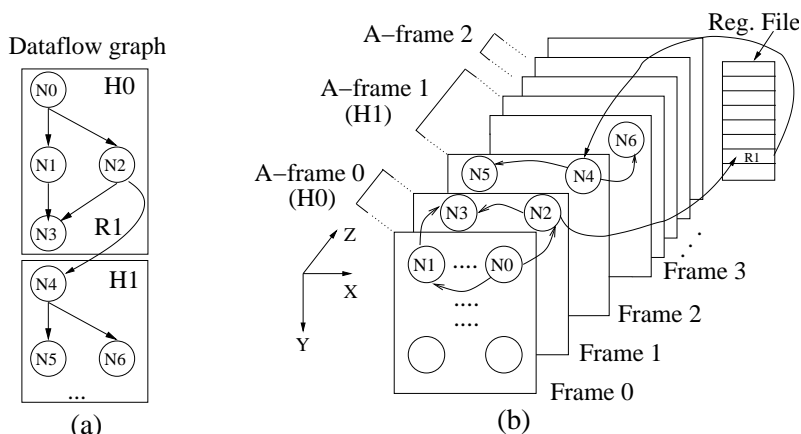


Fig. 4. D-morph frame management.

### 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 thus has similarities to previous work [Nagarajan et al. 2001], but also 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, using the TRIPS EDGE 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 beyond current-generation issue window sizes are possible. This window is logically 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 3c. This three-dimensional region can be viewed as a series of *frames*, as shown in Figure 4b, in which each frame consists 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 [Mahlke et al. 1992]. 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*. For example, a machine with 64 frames might be subdivided into 8 A-frames of 8 frames each, thus permitting 8 hyperblocks to be mapped concurrently. Within an A-frame, all relative positioning of instructions is statically determined, since a hyperblock fitting in one A-frame is the compiler target. Which of the multiple A-frames a hyperblock fits into is dynamically determined at run time, so dynamic naming support is needed when operands are transmitted across hyperblock boundaries. Communication within a hyperblock simply uses the same A-frame ID for the source and the destination.

Figure 4a shows a four-instruction hyperblock (H0) mapped into A-frame 0 as shown in Figure 4b, where instructions 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. Consumer targets are encoded in the producer instructions as previously described in this paper and elsewhere [Nagarajan et al. 2001]. 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 8, thus requiring 64 frames total in each processor core.

### 3.2 Multiblock Speculation

The TRIPS instruction window size is much larger than the average hyperblock size that can be constructed—a 1024 entry window in the prototype. In the D-morph, 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, similar to tasks in a Multiscalar processor [Sohi et al. 1995]. The A-frames differ from Multiscalar tasks in that they are contiguous in program order, whereas Multiscalar tasks each had their own instruction sequencer and were non-contiguous.

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 (the frame identifier portion of the target) of its target field to form the correct instruction buffer address of the consumer. Values passed between different hyperblocks are transmitted through the register file, as shown by the communication of R1 from H0 to H1 in Figure 4b. Such values are aggressively forwarded when they are produced, using a register stitch table that dynamically matches the register outputs of earlier hyperblocks to the register inputs of later hyperblocks. When a block is fetched from the instruction cache, the register stitch table logs the block's register inputs and outputs. For each input register, the

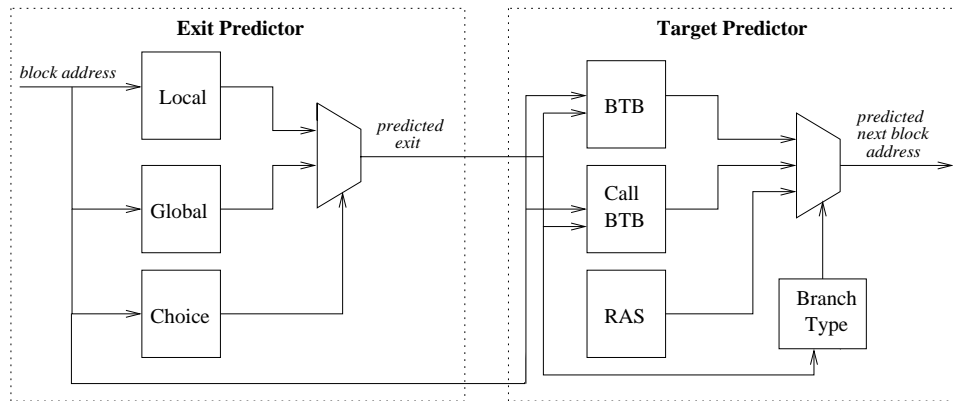


Fig. 5. TRIPS Exit Predictor and BTB.

register fetch logic reads the register stitch table to determine if the register will be produced by a previous block or if the stable value should be fetched from the non-speculative register file. When a register value is produced and delivered to the register file, the register stitch logic automatically forwards the value to the ALUs holding instructions from subsequent blocks that are waiting for it.

### 3.3 High-Bandwidth Instruction Fetching

To fill the large distributed window, the D-morph mode requires a high-bandwidth instruction fetch engine. The control model uses a program counter that points to hyperblock headers only. Hyperblocks are encoded as fixed-size instruction blocks, similar to VLIW instructions (which contain multiple instructions in a fixed format), but with no implicit ordering of instruction execution other than data dependences. Prepend to each block is a header that contains the number of stores, register outputs, and branches in each block.

When there are sufficient available frames to map the next hyperblock, the control logic makes a hyperblock prediction, generates the index, checks the instruction cache tags, and upon a hit accesses the 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. The instruction cache partitioning and broadcast of only the index multiplies the effective bandwidth that can be obtained from the I-cache, benefiting from the compiler-orchestrated layout of hyperblocks.

The next-hyperblock prediction is made using a scaled-up tournament exit predictor [Jacobson et al. 1997], which predicts a binary value indicating the branch that is predicted to be the exit of the hyperblock. Figure 5 shows the predictor itself, in which the value generated by the exit predictor is used to index into a set of BTBs to obtain the next predicted hyperblock address. The branch type is also predicted by the exit predictor, and is used to select an address from the multiple BTBs. We describe the predictor in further detail elsewhere [Ranganathan et al. 2002]. This predictor organization exploits the restriction that each block emits one

and only one branch thus avoiding the need to scan the instructions to make the prediction, which permits the predictor to be decoupled from the instruction fetch engine. The per-block accuracy of the exit predictor is shown in row 3 of Table I across a subset of the SPEC benchmark suite.

Although we do not simulate it in the performance analysis, the prototype will contain the capability to *revitalize* blocks, avoiding the re-fetch of blocks when the predicted block address matches the oldest block being removed from the execution substrate. With this capability, aligned loops can persist stably on the execution array until loop completion, saving the energy of instruction fetching, while requiring no software support to do so. We describe this feature in more detail in Section 5.

### 3.4 Memory Interface

To support high ILP, the D-morph memory system must provide both a high-bandwidth, low-latency data cache, and maintenance of sequential memory semantics. As shown in Figure 3b, the right side of each TRIPS core contains distributed primary memory system banks, which 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 simultaneously.

Each bank is coupled with MSHRs for the cache bank and a copy of the replicated load/store queues (LSQ), all of which use the same interleaving scheme. The TRIPS prototype uses the distributed LSQs to enforce ordering of loads and stores, restricting the number of loads and/or stores per block (32 per 128-instruction blocks) to keep the queue sizes tractable. The LSQs also use dependence prediction for aggressive out-of-order issue of loads while older store addresses are unresolved.

In the simulation results presented in this paper, however, we assume optimistic oracular load/store ordering, in which loads are issued as soon as it is correct to do so. Some of our current work in selective re-execution has recently shown that realistic load/store queues achieve 79% of an oracle, on average, and that gap continues to close with further research. We have also shown how to reduce the number of accesses and therefore energy consume in large LSQs [Sethumadhavan et al. 2003]. Since this supplemental work has shown how to make memory ordering scalable from both power and performance perspectives in TRIPS-like processors, the memory ordering assumptions in this paper's simulations are optimistic but reasonable.

The secondary memory system in the D-morph configures the networked banks as a non-uniform cache access (NUCA) array [Kim et al. 2002]. This network also provides a high-bandwidth link to each L1 bank for parallel L1 miss processing and fills. 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 polymorphous instruction buffers as an effective, distributed out-of-order issue window, demonstrated by the performance results in the next subsection.

### 3.5 D-morph Results

This subsection shows the measured ILP achieved using the D-morph mechanisms described above. The results shown in this section assume a slightly more powerful

Benchmark	adpcm	ampp	art	bzip2	compress	dct	equake
Good insts/block	30.7	119	80.4	55.8	21.6	163	33.5
Avg. frames	4.3	9.7	6.0	5.0	2.3	11.8	3.3
Exit/target pred. acc.	0.72	0.94	0.99	0.74	0.84	0.99	0.97
Insts in window	116	1126	1706	364	129	1738	622
Benchmark	gzip	hydro2d	m88k	mcf	mgrid	mpeg2	parser
Good insts/block	36.2	200	40.2	29.8	179	81.3	14.6
Avg. frames	5.5	14.5	4.0	4.0	12.8	7.0	1.9
Exit/target pred. acc.	0.84	0.97	0.95	0.91	0.99	0.88	0.93
Insts in window	671	1573	796	462	1590	958	255
Benchmark	swim	tomcatv	turb3d	twolf	vortex	mean	
Good insts/block	361	210	160	48.9	29.4	99.8	
Avg. frames	22.9	14.4	12.1	4.7	3.2	7.9	
Exit/target pred. acc.	0.99	0.98	0.94	0.76	0.99	0.91	
Insts in window	1928	1629	1399	361	918	965	

Table I. Execution characteristics of D-morph codes.

configuration of TRIPS processor than will be implemented in the prototype. These results 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 of wire delay 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 [V.Kathail et al. 2000], which is based on the Illinois Impact compiler [Chang et al. 1991]. The binaries were scheduled for the TRIPS processor with a custom scheduler/rewriter and converted into a high-level ISA that approximates the TRIPS prototype ISA. The compiler for the actual TRIPS ISA was not yet stable at the time of this writing.

The benchmarks used for the D-morph and T-morph are the subset of the Spec2000 Integer and Floating-point suites that could be compiled using the Trimaran tool set. For both the TRIPS and Alpha ISA experiments, we used SimPoint to determine a 200 million instruction region to execute each benchmark, and ensured that this region was beyond the initialization phases [Sherwood et al. 2002], which entailed fast forwarding between one and seven billion instructions for different benchmarks. We present results measured in Instruction per Cycle (IPC), in which we count only useful instructions, discounting overhead instructions associated with moving data from place to place, instructions with false predicates, and instructions not executed because of a block exit. In Section 5, to be more consistent with vector processor-style performance evaluation, we use a slightly different metric of computation instruction per cycle, which further discounts loads, stores, branches, and address computation instructions.

The first row of Table I shows the average number of useful dynamically executed instructions per block. The second row shows the average dynamic number of frames allocated per block by the TRIPS scheduler for a 4x4 ALU array. By computing the number of non-mispredicted hyperblocks using the steady-state block

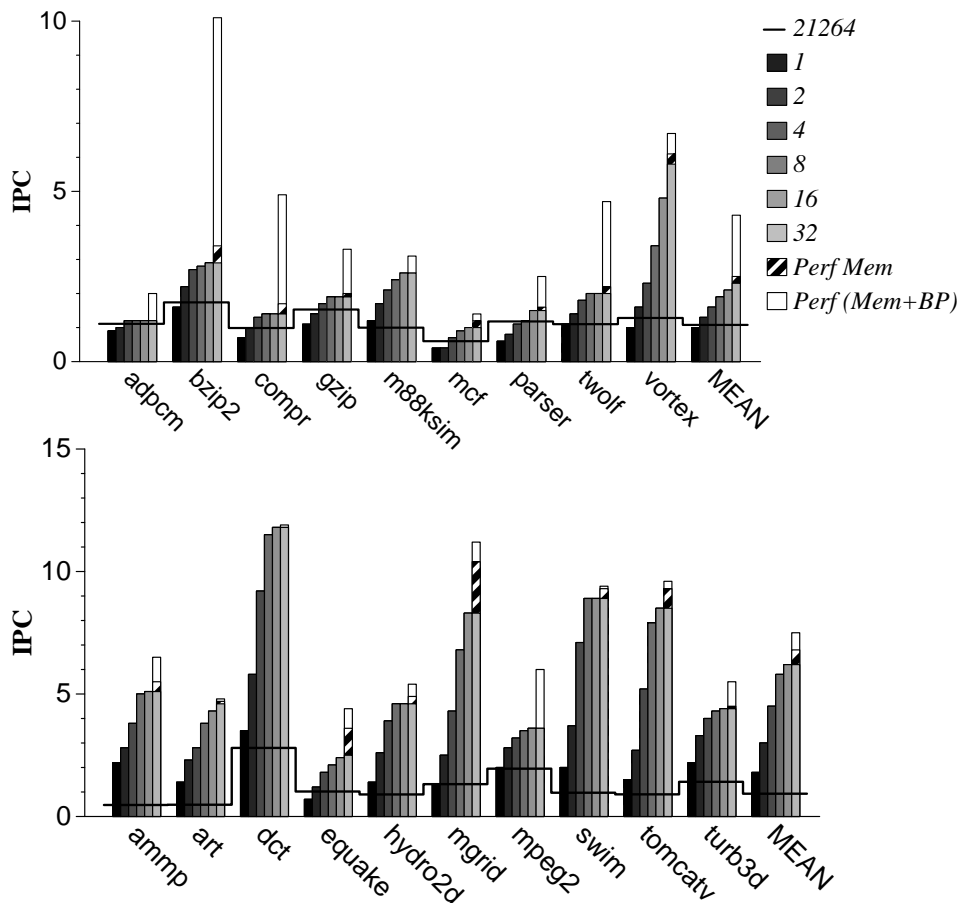


Fig. 6. D-morph performance as a function of A-frame count.

(exit) prediction accuracies shown in the third row, and the number of useful instructions per hyperblock (or frame), we can compute the mean number of useful instructions residing in the distributed window at any point in time, which is 965 instructions across the benchmark suite, as we show in row 4 of Table I.

Figure 6 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 top; the floating point and Mediabench [Lee et al. 1997] benchmarks are shown on the bottom. 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 benchmarks or saturation of the ALU array 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*.

The horizontal line across each set of bars represents the IPC of an Alpha 21264 for SimPoint regions similar to those run in the TRIPS experiments. For the integer benchmarks—with the maximum number of frames that we simulated—the IPC gain of TRIPS over the Alpha ranges from negligible (*adpcm*) to six-fold (*vortex*), but on average is a factor of two higher. With perfect branch prediction, the TRIPS IPC is nearly four times higher than the Alpha, due to the greater number of instructions found in the large TRIPS window. For the floating-point benchmarks, the TRIPS IPC for 32 hyperblocks is on average six times greater than the Alpha, due to the greater number of ALUs available in the TRIPS microarchitecture. While these performance gains are significant, enabling other modes of execution will increase the gains even more, as we see in subsequent sections.

#### 4. T-MORPH: THREAD-LEVEL PARALLELISM

When available single-thread parallelism is low, 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 [Tullsen et al. 1995] in that the execution resources (ALUs) and memory banks are shared, the T-morph statically partitions the reservation station (issue window) and eliminates the replicated reorder buffer required by SMT processors.

##### 4.1 T-Morph Implementation

There are multiple strategies for partitioning a TRIPS core to support multiple threads, two of which are *row threads* and *frame threads*. Row threads 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 threads, evaluated in this paper, time-share the processor by allocating threads to unique sets of physical frames. Below, we describe the polymorphous capabilities required to support multiple threads.

**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—in which threads can consume varied fractions of the frame space—is also possible, but is not implemented in the TRIPS prototype. Within each thread, the frames are further divided into some number of A-frames, permitting speculative execution within each thread. No additional register buffering is required, since the same storage used to hold state for speculative blocks can instead store state from multiple non-speculative and speculative blocks. However a separate copy of the architectural register file—128 architectural registers in the prototype—is required for each thread. The only addi-

Benchmarks	Throughput (aggregate IPC)			Norm. tpt.Eff. (%)	Speedup
	T-morph	Constant A-frames	Scaled A-frames		
<b>2 Threads</b>					
<i>bzip<sub>l</sub>, m88ksim<sub>l</sub></i>	4.9	5.5	5.5	90	1.8
<i>parser<sub>l</sub>, m88ksim<sub>l</sub></i>	3.7	3.8	4.1	90	1.8
<i>art<sub>h</sub>, compress<sub>l</sub></i>	5.1	5.7	6.0	86	1.6
<i>mc<sub>f</sub><sub>h</sub>, bzip<sub>l</sub></i>	3.2	3.9	3.9	81	1.7
<i>art<sub>h</sub>, mc<sub>f</sub><sub>h</sub></i>	5.1	5.3	5.6	90	1.8
<i>equake<sub>h</sub>, mc<sub>f</sub><sub>h</sub></i>	3.3	3.4	3.5	95	1.8
MEAN	4.7	-	-	87	1.7
<b>4 Threads</b>					
<i>bzip<sub>l</sub>, m88ksim<sub>l</sub>, parser<sub>l</sub>, compress<sub>l</sub></i>	6.1	6.7	8.4	72	2.9
<i>equake<sub>h</sub>, art<sub>h</sub>, parser<sub>l</sub>, compress<sub>l</sub></i>	6.1	7.0	10.0	61	2.2
<i>tomcatv<sub>h</sub>, mc<sub>f</sub><sub>h</sub>, m88ksim<sub>l</sub>, bzip<sub>l</sub></i>	8.3	10.7	15.0	55	2.3
<i>equake<sub>h</sub>, art<sub>h</sub>, tomcatv<sub>h</sub>, mc<sub>f</sub><sub>h</sub></i>	9.0	10.5	16.6	54	2.2
MEAN	7.4	-	-	61	2.4
<b>8 Threads</b>					
<i>art, tomcatv, bzip, m88ksim</i> <i>equake, parser, compress, mc<sub>f</sub></i>	9.8	17.7	25.0	39	2.9

Table II. T-morph thread efficiency and throughput.

tional frame supports needed are 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. Also, multiple return/address stacks are needed to prevent inter-thread interference in return target prediction.

**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 selected eight different benchmarks and ran different combinations of 2, 4 and 8 benchmarks executing concurrently. The high memory intensive benchmarks are *art<sub>h</sub>*, *mc<sub>f</sub><sub>h</sub>*, *equake<sub>h</sub>*, and *tomcatv<sub>h</sub>*. The low memory intensive benchmarks are *compress<sub>l</sub>*, *bzip<sub>l</sub>*, *parser<sub>l</sub>*, and *m88ksim<sub>l</sub>*. 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 ALU array, *b*) cache pollution, *c*) pollution and interaction in the branch predictor tables, and *d*) reduced speculation depth for each thread,

Benchmarks	T-morph	Perf. ALU+net	Perf. D-cache	Perf. I-fetch	D-morph
<i>bzip<sub>l</sub>, m88ksim<sub>l</sub></i>	4.9	5.3	4.9	5.0	5.5
<i>art<sub>h</sub>, compress<sub>l</sub></i>	5.1	5.1	5.5	5.0	5.7
<i>mcf<sub>h</sub>, bzip<sub>l</sub></i>	3.2	3.2	3.9	3.2	3.9
4 Threads					
<i>bzip<sub>l</sub>, m88ksim<sub>l</sub>, parser<sub>l</sub>, compress<sub>l</sub></i>	6.1	6.8	6.4	6.2	6.7
<i>equake<sub>h</sub>, art<sub>h</sub>, parser<sub>l</sub>, compress<sub>l</sub></i>	6.1	6.4	6.9	6.0	7.0
<i>tomcatv<sub>h</sub>, mcf<sub>h</sub>, m88ksim<sub>l</sub>, bzip<sub>l</sub></i>	8.3	9.5	8.5	8.4	10.7
<i>equake<sub>h</sub>, art<sub>h</sub>, tomcatv<sub>h</sub>, mcf<sub>h</sub></i>	9.0	10.0	9.1	9.0	10.5
8 Threads					
<i>art, tomcatv, bzip, m88ksim equake, parser, compress, mcf</i>	9.8	11.7	10.4	9.9	17.7

Table III. T-morph sources of performance loss.

since the number of available frames for each thread is reduced.

Table II 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 the same number of frames as available to each thread in the T-morph. A comparison of 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. A comparison of this column with column 2 indicates the performance drop incurred from both inter-thread interaction and reduced speculation in the T-morph. Our experiments showed that T-morph performance is largely insensitive to cache and branch predictor pollution.

Column 5 shows the normalized throughput as a ratio of T-morph throughput to that achievable by executing each thread on its own processor (column 2/column 4). This column also implies the approximate average slowdown seen by each thread when running in T-morph mode. With up to four threads, the per-thread slowdown is limited to 30-45%, but with 8 threads, it increases to 60%. 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. 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; the pairing of high memory benchmarks is 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.

Table III shows the sources of performance loss for different combinations of

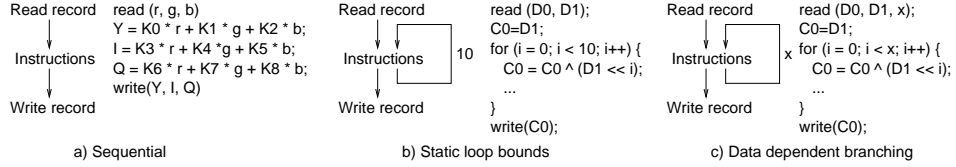


Fig. 7. Data parallel kernel control behavior.

threads. Column 1 shows the T-morph performance as in Table II. Columns 2 through 4 respectively show the T-morph IPC assuming no ALU or network contention, private per-thread D-caches of the same size, and private I-cache and instruction distribution networks, respectively. For each of these three experiments, the component is simulated as if each thread had the same resources (D-cache space, I-fetch bandwidth, etc.) as if the processor were running a single thread. In the rightmost column, the table displays the sum of the instruction throughputs assuming separate processors, but assumes that each thread only has a fraction of the instruction window, as in the constant A-frame experiment above. This number represents the upper bound of T-morph performance.

For two threads, D-cache interference typically plays the strongest role. For four and eight threads, ALU/network contention causes the largest performance drop (comparing column 6 against column 3), although the shared D-cache does cause a large performance drop for the second 4-thread experiment.

## 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 [Talla et al. 2003], 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 [Rixner et al. 1998].

### 5.1 Data Parallel Application Characteristics

Data-parallel workloads can be classified into domains based on the type of data being processed. The nature of computation varies within a domain and across the different domains. The applications vary from simple computations on image data converting one color space to another (comprising 10s of instructions) to complex encryption routines on network packets (comprising 100s of instructions). Four broad categories cover a significant part of this spectrum: digital signal processing, network/security, scientific, and real-time graphics. We examine the specific characteristics of applications from these domains, categorized by their effect on the memory system, instruction control, and processor execution core.

The **memory behavior** of data-parallel applications can be classified into four different types: (1) regular memory accesses in which memory is read in a structured and predictable manner, (2) irregular memory accesses which consist of un-

Benchmark	Description
<i>Multimedia processing</i>	
convert	RGB to YIQ conversion
dct	A 2D DCT of an 8x8 image block
highpassfilter	A 2D high pass filter
fir16	Finite impulse response filter with 16 taps
transform	Transformation of vertices in 3-D space
<i>Network processing, security (1500 byte packets)</i>	
md5	MD5 checksum
rijndael	Rijndael (AES) packet encryption
blowfish	Blowfish packet encryption
idea	Encryption of a 128-bit block (used in PGP)
<i>Scientific codes</i>	
fft8	8-point complex Fast Fourier Transform
lu	LU decomposition of a dense 1024x1024 matrix
<i>Real-time graphics processing [Fernando and Kilgard 2003]</i>	
vertex-simple	Vertex lighting with ambient, diffuse, specular and emissive
fragment-simple	Fragment lighting with ambient, diffuse, specular and emissive
vertex-reflection	Vertex shader for a reflective surface
fragment-reflection	Fragment shader rendering a reflective surface using cube maps
vertex-skinning	A vertex shader for animation with multiple transformation matrices
anisotropic-filtering	Fragment shader implementing anisotropic texture filtering

Table IV. Benchmark description.

predictable memory addresses, such as those found with texture mapping tables in a graphics system, (3) named constant scalar operands, such as coefficients in filtering kernels that do not change during execution, and (4) indexed constant operands such as small lookup tables like those used for encryption kernels. In characterizing DLP programs, we are interested in the frequency of occurrence of each of the four types of accesses in a kernel. The four types of accesses are not exclusive; a kernel may use all four categories.

The complexity of the **control structure** within the inner loops determines the type of synchronization and instruction sequencing required. As shown in Figure 7a, the simplest kernels contain a sequence of instructions with no internal control flow. A degenerate case is a single vector operation, but the 2D DCT can be transformed into this model by unrolling all of the internal computations of the 8x8 kernel. Each iteration of these kernels executes in the exact same fashion, so these kernels are well-suited for vector or SIMD control. A slightly more complex type of control behavior occurs when the kernel contains loops with static loop bounds. Figure 7b shows this type of control behavior with an example encryption kernel pseudo-code. Such kernels can be unrolled at compile time increasing the code size of the kernel, although for some kernels this transformation results in prohibitively large instruction storage requirements. Architectures that lack any branching support (like some graphics fragment processors) must rely on complete unrolling to execute such loops. Figure 7c shows the most generic of control behavior: data dependent branching. These kernels with runtime determined loop bounds would require masking instructions to execute on vector and SIMD machines, and are ideally suited to fine-grain MIMD machines, since each processing element can be independently controlled according to the local branching behavior.

Benchmark	Computation		Memory				Control
	# inst	ILP	rec. size r/w	# irr. mem. acc.	# const	# Indexed scalar constants	Loop bounds
convert	15	5	3/3	-	9	-	-
dct	1728	6	64/64	-	10	-	16
highpassfilter	17	3.4	9/1	-	9	-	-
fir16	34	8.5	1/1	-	16	-	-
transform	37	9.2	8/8	-	21	-	-
md5	680	1.63	10/2	-	65	-	-
blowfish	364	1.98	1/1	-	2	256	16
rijndael	650	11.8	2/2	-	18	1024	10
idea	112	2.2	2/2	-	54	-	-
fft8	104	20.8	16/16	-	16	-	-
lu	2	1	2/1	-	0	-	-
vertex-simple	95	4.3	7/6	-	32	-	-
fragment-simple	64	2.96	8/4	4	16	-	-
vertex-reflection	94	7.1	9/2	-	35	-	-
fragment-reflection	98	6.2	5/3	4	7	-	-
vertex-skinning	112	6.8	16/9	-	32	288	Variable
anisotropic-filter	80	2.1	9/1	≤ 50	6	128	Variable

Table V. Benchmark attributes with record size in 32-bit words.

Table IV describes a suite of DLP kernels selected from four major application domains. Table V characterizes these kernels according to the computation, memory and control criteria already presented. The two computation columns list the number of instructions and inherent ILP within the kernel. Here, ILP is computed as the number of instructions in one iteration of a kernel divided by the dataflow graph height; when the loop bound is variable, the kernel is completely unrolled. The first memory column lists the size of the record (in 64-bit words) that each kernel reads and writes, the second column gives the number of irregular memory accesses, and the third and fourth memory columns describe the use of static coefficients within the kernel and the size of the lookup table for indexed constants, if one is needed. The control column indicates the number of loop iterations within the kernel (if any) and whether the loop bounds are variable across kernel instances, in which case the kernels exhibit data dependent control and are well-matched to a fine grained MIMD execution model. In the *anisotropic-filter* kernel, for example, the number of instructions executed varies from about 150 to 1000 for each instance. In vector or SIMD architectures, which lack support for fine grain branching, each instance would execute all 1000 instructions, using predication or other techniques for nullifying unwanted instructions. Collectively, the benchmarks exhibit wide variation in each of the attributes, demonstrating diversity in the fundamental behavior of DLP applications. We used this application study identify and examine attributes and complementary microarchitectural mechanisms for the S-morph [Sankaralingam et al. 2003].

## 5.2 S-morph Mechanisms

The S-morph was heavily influenced by the Imagine architecture [Khailany et al. 2001] and uses the Imagine execution model in which a set of stream kernels are

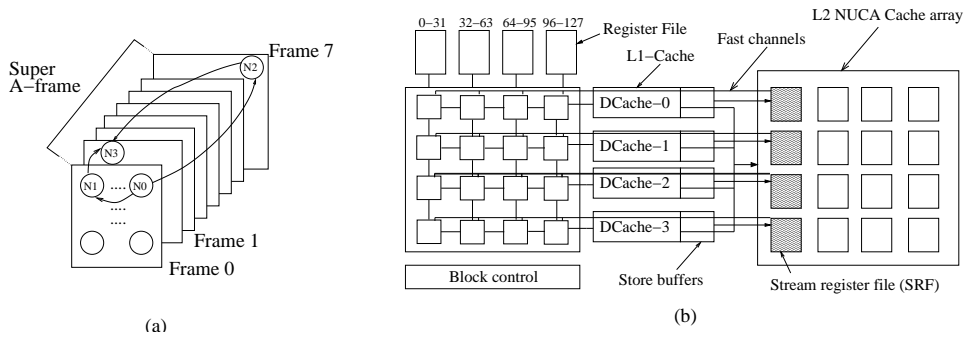


Fig. 8. Polymorphism for S-morph.

sequenced by a control thread. Figure 8 highlights the features of the TRIPS processor which are configured for data parallel applications in the S-morph.

**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. To avoid increasing the instruction length to accommodate a larger number of named reservation stations, the S-morph naming scheme can be adjusted so that each instruction specifies a destination ALU relative to the current ALU, rather than an absolute position. 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 as 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 implements mapping reuse with a `<repeat N>` instruction (similar to RPTB in the TMS320C54x [TMS320C54x]) 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 the reservation stations 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 [Mai et al. 2000], 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

Benchmark	Fused Iterations				# of Revitalizations
	Unrolling factor	Compute insts per block	Block size	Total Constants	
convert	16	240	303	144	171
dct	8	560	580	80	128
fir16	16	544	620	256	512
fft8	4	416	570	64	128
idea	8	896	1020	416	512
transform	16	592	740	336	64

Table VI. Characteristics of S-morph codes.

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 8b, 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* (LMW) instruction by transferring an entire SRF line into the ALU array, spreading it across the ALUs in a fixed pattern within a row. Once within the ALU array, 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.

### 5.3 Results

We evaluate the performance of the TRIPS S-morph on a subset of streaming kernels, as shown in Table VI, which were extracted from the Mediabench benchmark suite [Lee et al. 1997]. 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:** Table VI shows the intrinsic characteristics of the kernel after being unrolled to fill the A-frames. 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” metric includes only computation instructions while the block size numbers include overhead instructions for memory access and data movement within the ALU array. The total constant count indicates the number of reservation stations that must be

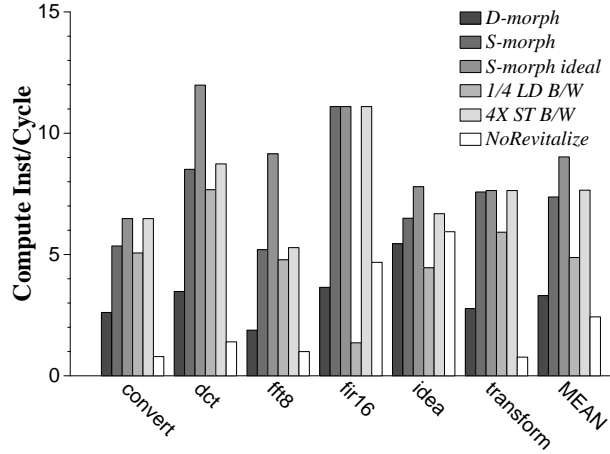


Fig. 9. S-morph performance.

filled with constant values from the register file for each iteration of the unrolled loop. Most 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 assumes on 64-Kbyte input and output streams, which are both striped across and stored in the SRF.

**Performance analysis:** Figure 9 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 9 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 [Espasa et al. 2002] 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 array consisting of 32 ALUs sustains, on average, 15 compute ops per cycle.

**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 9, 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 negative effects 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.

#### 5.4 S-morph Summary

The most effective augmentations to the TRIPS processor to accelerate data parallel applications are (1) revitalization to improve instruction fetch and (2) the high bandwidth channels—coupled with a load multiple word instruction—to improve data fetch bandwidth. The SRF structure is essentially a software managed cache, which can be implemented directly using a dedicated memory, or emulated using a larger hardware managed cache with software support for prefetching and eviction hints. Similar augmentations can be applied to other architectures to make them better suited to DLP programs. The streaming register file SRF, store buffer, and the LMW instructions can be added in a straightforward manner to conventional wide-issue centralized or clustered superscalar architectures by adding direct channels from the L2 caches to the functional units and augmenting the pipeline to wakeup instructions dependent on the loads when their operands arrive from the SRF. The Tarantula architecture provides similar such support for transfers from the L2 memory to the vector register file, using hardware techniques to generate conflict-free addresses to different banks in memory, in contrast to the approach of packing all of the regular accesses into a single bank. To support indexed scalar accesses and irregular memory accesses in this architecture, the L1 cache memory must be addressable using special scatter/gather instructions. In this article, we only evaluate applications with sequential control behavior on the S-morph, which requires relatively minimal enhancements beyond the D-morph. However, other applications have even more diverse demands on the program control mechanisms of data parallel processors. For example, to better exploit irregular data parallelism requires less rigid control as provided by fine-grain MIMD architectures. An examination and evaluation of hybrid SIMD/MIMD style mechanisms for TRIPS can be found in [Sankaralingam et al. 2003].

## 6. TRIPS SCALABILITY

While the experiments in Sections 3–5 reflect the performance achievable on three distinct application classes with 16 ALUs, the question of granularity still remains.

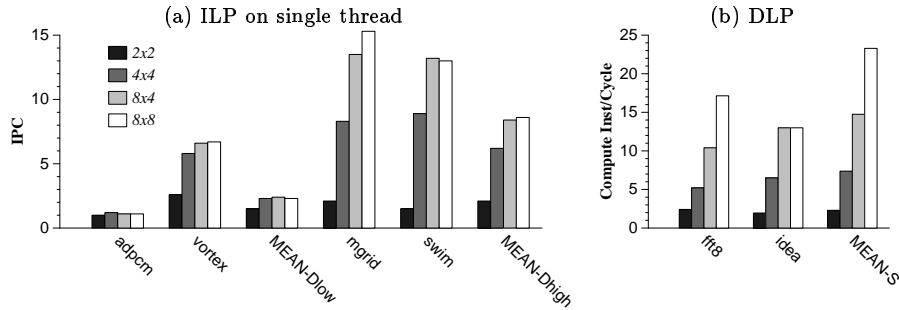


Fig. 10. TRIPS single-core scalability.

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—and thus size—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.

### 6.1 Current Optimal Core Sizes

Figures 10a and 10b 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 numbers of ALUs. Table VII 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 VII shows several alternative TRIPS chip designs, ranging from 8 2x2 TRIPS cores to 2 8x8 cores, assuming a  $400\text{mm}^2$  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 11 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 a large number of threads is 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

Grid Dimensions	Preferred Applications	# TRIPS cores per chip	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 VII. TRIPS CMP Designs.

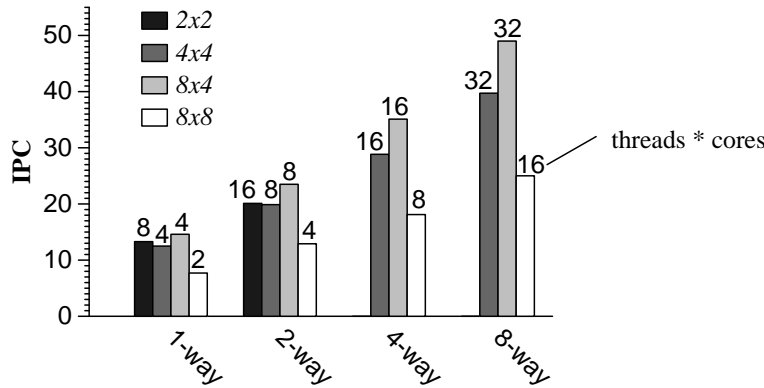


Fig. 11. TRIPS CMP throughput with multiple threads.

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.

## 6.2 Scaling to Larger Cores

Scaling to cores even larger than those measured in this paper faces several challenges. First, providing sufficient instruction and data bandwidth into the array becomes difficult at large array sizes. Solving the instruction bandwidth problem is not difficult; since block mappings into the instruction cache are statically managed, the instruction cache banks can easily be distributed into the array, requiring only the broadcast of an i-cache index to all nodes for each block fetch. Distributing the data caches, however, is problematic, as coherence and/or sequential memory semantics become difficult to enforce. Much progress on memory disambiguation must be made before such fully distributed data caches are feasible.

The other major challenge for irregular codes is control predictability. As the array size increases, the instruction window size increases proportionately. However, if there are next-block mispredictions in flight, the larger window will be underutilized, resulting in longer communications when a smaller array with a smaller window and smaller routing distances would be a better match. We are currently exploring aggressive predication techniques to construct hyperblocks that end on control flow graph merge points, improving predictability sufficiently well to exploit larger array sizes.

## 7. RELATED WORK

The TRIPS architecture achieves its performance and scalability by exploiting three major characteristics of the hardware/software interface. First, like VLIW architectures [Rau and Fisher 1993], the placement of instructions onto ALUs is encoded into the instruction stream. This mechanism eliminates the need for hardware placement algorithms and reduces the burden on bypassing networks as operation destinations are encoded in the instructions. Second, operands are not just sent to the target ALU, they are sent directly to a target instruction at the target ALU in a point-to-point fashion, with similarities to tagged token dataflow architectures [Arvind and Nikhil 1990]. Encoding the target ALUs through the use of reservation stations enables elimination of associative wakeup and selection logic. Finally, the TRIPS execution model is a hybrid between von-Neumann and dataflow architectures, with sequential ordering for hyperblock fetch and mapping but dataflow execution ordering within the block. Dynamic links between blocks permit concurrent execution and a larger effective dataflow graph. This aggregation of instructions into atomic blocks is similar to prior work examining large atomic units [Melvin and Patt 1989; Hao et al. 1996] as well as other hybrid von Neumann/dataflow approaches [Hwu and Patt 1986], although it pushes the size of executed graphs much farther.

We term the combination of these three mechanisms in the instruction set architecture the explicit data graph execution (EDGE) instruction set. The prior architectures mentioned above have some attributes of EDGE instruction sets, but reside in a different part of the space of instruction sets. The Wavescalar architecture is perhaps the most similar to TRIPS and could be interpreted as having an EDGE instruction set [Swanson et al. 2003] because of the placement and target encoding directives. The Wavescalar ISA uses mechanisms quite similar to TRIPS to order loads and stores, with IDs that have a dynamically assigned portion and a statically specified tag within hyperblocks (or waves).

The TRIPS goals of scalability through partitioning has been examined by many prior architectures, including statically scheduled VLIW systems, dynamically scheduled superscalar processors [Kessler 1999; Ranganathan and Franklin 1998; Kim and Smith 2002], and hybrids such as Multiscalar [Sohi et al. 1995]. The TRIPS architecture pushes partitioning further, exposing a larger and finer-grained partitionable substrate directly to large blocks in the instruction set for optimization by the compiler.

One of the other goals of the TRIPS architecture is the ability to adapt to many different application types. Many previous architectures have been configurable

and even dynamically adaptive. Beyond the examples of reconfigurable hardware described in Section 1, several architectures have recently emerged to adapt the hardware to changing dynamic demands. Complexity-adaptive architectures can dynamically trade off capacity and delay of on-chip memory structures such as caches [Albonesi 1998]. Other approaches dynamically modulate the clock frequency and voltage to simultaneously optimize for power and performance [Semeraro et al. 2002; Li et al. 2003]. Because TRIPS presents a different view of the hardware to the operating system and compiler depending on the selected execution mode, we term it polymorphous rather than configurable or adaptive.

## 8. CONCLUSIONS AND FUTURE DIRECTIONS

The polymorphous TRIPS processor 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, enabled by the TRIPS EDGE ISA. The goal of this system is to achieve performance and efficiency approaching that of special-purpose systems. In this paper, we have 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 this space as a part of the development of the TRIPS prototype system.

## REFERENCES

- ALBONESI, D. 1998. Dynamic ipc/clock rate optimization. In *25th International Symposium on Computer Architecture*. 282–292.
- ARVIND AND NIKHIL, R. S. 1990. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computing* 39, 3 (March), 300–318.
- BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B.,  
ACM Journal Name, Vol. V, No. N, Month 20YY.

- SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 282–293.
- BAUMGARTE, V., MAY, F., NÜCKEL, A., VORBACH, M., AND WEINHARDT, M. 2001. PACT XPP – A Self-Reconfigurable Data Processing Architecture. In *1st International Conference on Engineering of Reconfigurable Systems and Algorithms*.
- CASÇAVAL, C., CASTANOS, J., CEZE, L., DENNEAU, M., GUPTA, M., LIEBER, D., MOREIRA, J. E., STRAUSS, K., AND JR., H. S. W. 2002. Evaluation of multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*. 311–322.
- CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., WARTER, N. J., AND MEI W. HWU, W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 266–275.
- CINTRA, M., MARTÍNEZ, J. F., AND TORRELLAS, J. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 13–24.
- EBELING, C., CRONQUIST, D. C., AND FRANKLIN, P. 1997. Configurable computing: The catalyst for high-performance architectures. In *International Conference on Application-Specific Systems, Architectures, and Processors*. 364–372.
- ESPASA, R., ARDANAZ, F., EMER, J., FELIX, S., GAGO, J., GRAMUNT, R., HERNANDEZ, I., JUAN, T., LOWNEY, G., MATTINA, M., AND SEZNEC, A. 2002. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings ISCA-29*. 281–292.
- FERNANDO, R. AND KILGARD, M. J. 2003. *The Cg Tutorial*. Addison-Wesley Publishing Company.
- GOLDSTEIN, S. C., SCHMIT, H., BUDIU, M., CADAMBI, S., MOE, M., AND TAYLOR, R. 2000. Pipherench: A reconfigurable architecture and compiler. *IEEE Computer* 33, 4 (April), 70–77.
- HAO, E., CHANG, P., EVERS, M., AND PATT, Y. 1996. Increasing the Instruction Fetch Rate via Block-structured Instruction Set Architectures. In *Proceedings MICRO-29*. 191–200.
- HWU, W. AND PATT, Y. 1986. Hpsm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the International Symposium on Computer Architecture*. 297–306.
- JACOBSON, Q., BENNETT, S., SHARMA, N., AND SMITH, J. E. 1997. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*. 218–229.
- KANG, Y., HUANG, W., YOO, S.-M., KEEN, D., GE, Z., LAM, V., PATNAIK, P., AND TORRELLAS, J. 1999. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*. 192–201.
- KESSLER, R. 1999. The alpha 21264 microprocessor. *IEEE Micro* 19, 2 (March/April), 24–36.
- KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., MATTSON, P., NAMKOONG, J., OWENS, J. D., TOWLES, B., AND CHANG, A. 2001. Imagine: Media processing with streams. *IEEE Micro* 21, 2 (March/April), 35–46.
- KIM, C., BURGER, D., AND KECKLER, S. W. 2002. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings ASPLOS-10*. 211–222.
- KIM, H.-S. AND SMITH, J. E. 2002. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th International Symposium on Computer Architecture*. 71–82.
- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*. 330–335.
- LI, H., CHER, C.-Y., VIJAYKUMAR, T., AND ROY, K. 2003. Vsv: L2-miss-driven variable supply-voltage scaling for low power. In *36th Annual International Symposium on Microarchitecture*. 19–28.

- MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND BRINGMANN, R. A. 1992. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings MICRO-25*. 45–54.
- MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings ISCA-27*. 161–171.
- MELVIN, S. AND PATT, Y. 1989. Performance benefits of large execution atomic units in dynamically scheduled machines. In *3rd International Conference on Supercomputing*. 427–432.
- NAGARAJAN, R., SANKARALINGAM, K., BURGER, D., AND KECKLER, S. W. 2001. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings MICRO-34*. 40–51.
- OSKIN, M., CHONG, F. T., AND SHERWOOD, T. 1998. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th International Symposium on Computer Architecture*. 192–203.
- RANGANATHAN, N. AND FRANKLIN, M. 1998. The pews microarchitecture: Reducing complexity through data dependence based decentralization. *Microprocessors and Microsystems 22*, 6 (November), 333–343.
- RANGANATHAN, N., NAGARAJAN, R., BURGER, D., AND KECKLER, S. W. 2002. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Tech. Rep. TR-02-41, Department of Computer Sciences, The University of Texas at Austin. September.
- RAU, B. R. AND FISHER, J. A. 1993. Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing 7*, 9–50.
- RIXNER, S., DALLY, W. J., KAPASI, U. J., KHAILANY, B., LOPEZ-LAGUNAS, A., MATTSON, P. R., AND OWENS, J. D. 1998. A bandwidth-efficient architecture for media processing. In *Proceedings on the 31st International Symposium on Microarchitecture*. 3–13.
- SANKARALINGAM, K., KECKLER, S. W., MARK, W. R., AND BURGER, D. 2003. Universal Mechanisms for Data-Parallel Architectures. In *Proceedings MICRO-36*. 303–314.
- SEMERARO, G., ALBONESI, D., DROPSHO, S., MAGKLIS, G., DWARKADAS, S., AND SCOTT, M. 2002. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *35th International Symposium on Microarchitecture*. 356–367.
- SETHUMADHAVAN, S., DESIKAN, R., BURGER, D., MOORE, C. R., AND KECKLER, S. W. 2003. Scalable memory disambiguation for high ilp processors. In *36th International Symposium on Microarchitecture*. 399–410.
- SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.
- SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*. 414–425.
- STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 1–12.
- STERLING, T. L. AND ZIMA, H. P. 2002. The gilgamesh MIND processor-in-memory architecture for petaflops-scale computing. In *International Symposium on High Performance Computing*. 1–5.
- SWANSON, S., MICHELSON, K., SCHWERIN, A., AND OSKIN, M. 2003. Wavescalar. In *36th Annual International Symposium on Microarchitecture*. 291–302.
- TALLA, D., JOHN, L., AND BURGER, D. 2003. Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements. *IEEE Transactions on Computers 52*, 8, 35–46.
- TENDLER, J. M., DODSON, J. S., J. S. FIELDS, J., LE, H., AND SINHARROY, B. 2001. POWER4 system microarchitecture. *IBM Journal of Research and Development 26*, 1 (January), 5–26.
- TMS320C54X. DSP Reference Set, Volume 2: Mnemonic Instruction Set, Literature Number: SPRU172C, March 2001.
- TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings ISCA-22*. 392–403.

- V.KATHAIL, M.SCHLANSKER, AND B.R.RAU. 2000. Hpl-pd architecture specification: Version 1.1. Tech. Rep. HPL-93-80(R.1), Hewlett-Packard Laboratories. February.
- WAINGOLD, E., TAYLOR, M., SRIKRISHNA, D., SARKAR, V., LEE, W., LEE, V., KIM, J., FRANK, M., FINCH, P., BARUA, R., BABB, J., AMARASINGHE, S., AND AGARWAL, A. 1997. Baring it all to software: RAW machines. *IEEE Computer* 30, 9 (September), 86–93.
- XILINX. Virtex-II Pro X Platform FPGAs: Introduction and Overview, November 17, 2003.

Received Month Year; revised Month Year; accepted Month Year