

Aaron J Arthurs
Fault-Tolerant Systems
December 1, 2005
Class Project

Enhancing Reliability in the TRIPS Grid Array

Abstract:

This paper first introduces a reliability model that bases on the number of resources in the grid array used by a node for a certain instruction. A novel feature called the RX-morph enhances the reliability of the TRIPS architecture. RX-morph adds a new polymorphous resource for each of D-morph, T-morph, and S-morph that uses a form of time redundancy to reduce the length of data paths per frame.

Introduction:

A significant feature of threads is that threads maintain separate program contexts; that is, the logic treats each thread as separate programs that share a common pool of resources (e.g. shared memory). Many server applications make use of threads to divide various tasks into concurrent subroutines. For example, DBMS systems spawn transactions, which are atomic execution units that perform reads and writes to databases; eventually, each transaction will either commit its results and terminate or abort entirely. DBMS transactions execute as independent processes with a possibility of data dependencies. Another example is SSH servers that create threads to handle incoming and outgoing connections. It is clear that many servers can benefit from an architecture that exploits thread characteristics.

Thread level parallelism (TLP) is exploiting the fact that a system is running threads in order to execute tasks in parallel. More formally, TLP is the potential amount of thread concurrency that is present in a given system. On most architectures (including those that are TLP optimized), it is rare that a system's full TLP is achieved due to memory latencies, control logic, and mis-predictions. TLP is an extension of instruction level parallelism (ILP); in addition to executing instructions in parallel, multiple threads share the same CPU system in a concurrent manner. A unique property of TLP that ILP does not contain is the guarantee of program control independence between parallel instructions from different threads. In other words, each thread maintains its own program context, which can map directly into hardware via, e.g. an array of registers. Because of the program control independence, a CPU system can swap active threads in and out of execution and potentially reduce stalling due to program control instructions (e.g. branch). For instance, when an active thread is blocking for an I/O operation, the CPU can swap the blocking thread with a thread that is ready to execute. Thread swapping is one approach to improving CPU utilization.

Although the idea of TLP has been around for a while, designs that exploit TLP have recently begun emerging. One such design is the grid processor architecture (GPA) family of CPUs. GPAs consist of a two-dimensional array of processing elements surrounded by various memory banks and control logic. Each processing element contains an ALU, multiple instruction storage, and input/output routing control. The GPA family introduces dataflow ISAs that, unlike RISC ISAs, follow the statically placed, dynamically issued (SPDI) execution model. SPDI involves producer and consumer instructions such that data dependencies flow from the producers to the respective consumers, opposed to RISC architectures that maintain a central data structure shared

between instructions. In GPAs, instructions load into the processing element array and the interconnections between processing elements transfer data from producers to consumers.

TRIPS is an implementation of the GPA family that not only exploits TLP but also ILP and data level parallelism (DLP). It uses the idea of polymorphism to reconfigure the chip for one of the three parallelism schemes: D-morph, T-morph, and S-morph [5, 7]. D-morph (desktop-morph) utilizes ILP from single-threaded systems, T-morph (threaded-morph) utilizes TLP from multi-threaded systems, and S-morph (streaming-morph) utilizes DLP from large-data processing systems. In addition to the three morphs, RX-morph adds a new polymorphous resource to implement a reliable version of each morph. RX-morph (reliable D-, T-, and S- morph) bases on a reliability model, discussed in a later section, by shortening the data paths in the grid array at each time frame.

The following sections present the TRIPS architecture and prototype. Later, the paper discusses thread execution with an example to illustrate how data flows through the TRIPS grid array. Then, the next sections explain the reliability model used to measure the probability that a utilized node will operate correctly. Finally, the report lays out RX-morph's implementation and evaluates improvements in the reliability of each node.

The TRIPS Architecture:

Different applications have different needs, especially the need for a certain organization in a chip. There is a spectrum on the granularity of chips that ranges from ultra-fine grained to coarse grained. FPGAs typically fall under the ultra-fine grained category because of its highly configurable structure; these structures have many primitive components such as LUTs, ALU operations, and programmable interconnects. Fine-grained chips are suitable for parallel applications (e.g. threads) and applications with regular memory access. On the other extreme, coarse grained chips such as chip multiprocessors (CMP) work best with single-threaded applications with high ILP.

The TRIPS (Tera-op Reliable Intelligently adaptive Processing System) architecture adapts to a wide variety of applications by partitioning several coarse grained elements/cores into many fine grained structures. In addition to the cores are DRAM memory tiles that connect to the functional cores via the on-chip interconnection network (OCN). The memory tiles provides the system a non-uniform cache access (NUCA) L2 cache, scratch-pad memory, and synchronization buffers for producer to consumer communications. In the current TRIPS prototype (developed in 2004), there are two functional cores and an array of 32 64KB memory tiles connected by a routed network and distributed memory controllers with channels to external (main) memory. Each core contains a 4x4 array of processing elements surrounded by various memory banks and controllers. There is one register bank per column and data and instruction L1 cache per row. A block controller along side some prediction logic orchestrates the 4x4 array and the caches plus one additional instruction cache that stores block header information. Within each of the 16 processing elements are 64 storage slots for instructions, operand buffers, an ALU controlled by one of the instruction slots (referred to as slot 0), a node controller that directs input operands and the instruction slots, and a router that directs output data. The processing elements allow for out-of-order execution; that is, the system does not necessarily issue

instructions in the same order, as they would appear sequentially. As for the register banks, each bank stores 32 32-bit registers, making 128 registers per core. Along with the register banks is the register stitching logic contained in the block controller that controls communication via the registers.

Before explaining the mechanics of the TRIPS core structures, it is important to note that the TRIPS architecture implements a dataflow ISA called the explicit dataflow graph execution (EDGE) ISA. As with any dataflow ISAs, there are producer and consumer instructions; however, the EDGE ISA requires that a compiler store the target address in most instruction formats, which makes the dataflow visible to a compiler. One other job of a compiler targeting the TRIPS design is partitioning the software into blocks of instructions called hyperblocks.

Hyperblocks are atomic, meaning that the instructions contained in a hyperblock are acted upon

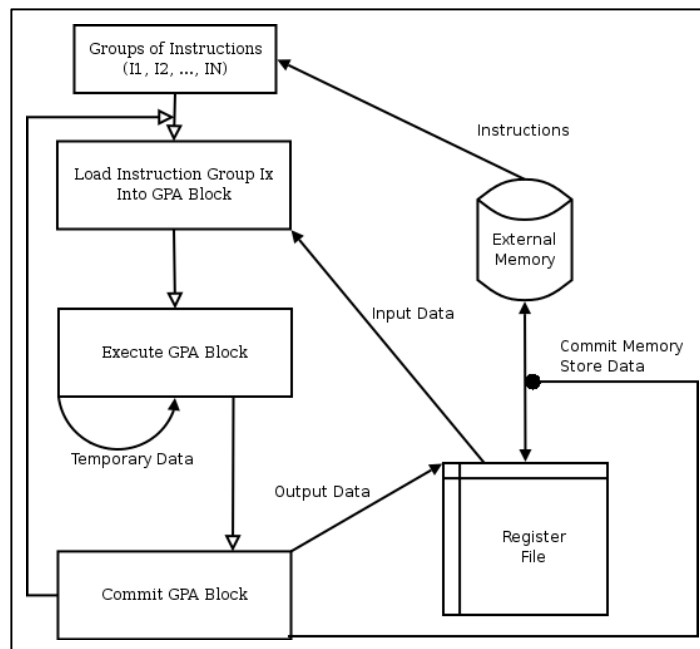


Figure 1: Block execution model

all together (i.e. the architecture does not extract instructions out of a hyperblock). Three requirements of hyperblocks are only one entry point, no internal loops, and at least one exit point. When a hyperblock is loaded into the chip, a register stitching logic forwards the input data from the output of a previous hyperblock or from the register file; it passes the values to the input-dependent nodes in each core. The hyperblock executes, passing temporary values between internal instructions using the node array interconnection. The hyperblock aborts, dropping any changes it may have made, or commits its output values to the register banks. After aborting or committing, the block controller will typically deallocate the hyperblock and begin execution of the next hyperblock. In the prototype, hyperblocks partition into five 128-byte chunks. The first chunk is the hyperblock header that stores what are the input and output registers. The other four chunks store 32 32-bit instructions each; each chunk maps to a row in the processing element array through the instruction L1 cache. Each row consists of four processing elements, and each

processing element stores eight instructions from a hyperblock. With 64 slots per processing element, eight hyperblocks can be stored in each core.

The instructions encode in one of six formats: Load, Store, Arithmetic, Arithmetic-Immediate, Generate-Constant, and Branch. Load, Arithmetic, Arithmetic-Immediate, and Generate-Constant instructions specify target consumer instructions. With the exception of Generate-Constant instructions, TRIPS can predicate each instruction. Predicated instructions conditionally execute depending on whether the predicate register at the given address in the instruction is set to true or false. Predication, along with early hyperblock exits, brings the issue of outputting the correct register values when there are predicated instructions producing outputs for the hyperblock [2]. Consequently, hyperblocks are restricted to output a single value per output register.

In what fashion the instructions execute depends on which morph the TRIPS chip is using. As mentioned earlier, TRIPS runs in D-morph, T-morph, and S-morph mode; T-morph is similar to D-morph except for certain program contexts such as the program counter. As in D-morph, T-morph uses a three-dimensional scheduling region that divides into two-dimensional frames. A frame is the aggregation of one instruction slot from each processing element. In the case of the TRIPS prototype, each frame holds 16 instructions from the 4x4 processing element array. Hyperblocks span across multiple frames, grouping contiguous sets of physical frames into architectural frames (A-frames). Because multiple hyperblocks map into each grid core, some hyperblocks may be speculative. A speculative hyperblock is a hyperblock loaded into a non-executing A-frame; speculation is observing the current state of the architecture and making predictions by pre-fetching certain data and instructions. At any given time, only one A-frame is active/non-speculative while the other A-frames are queued. A program counter points to the active A-frame.

T-morph Configuration:

Threads map into a core in several ways, two of which are row partitioning and frame partitioning. In row partitioning, a thread executes on one or more row in each core. The problem with this approach is that the distance from the register banks increases down towards the bottom rows. In other words, threads would have uneven register bank latencies. As for frame

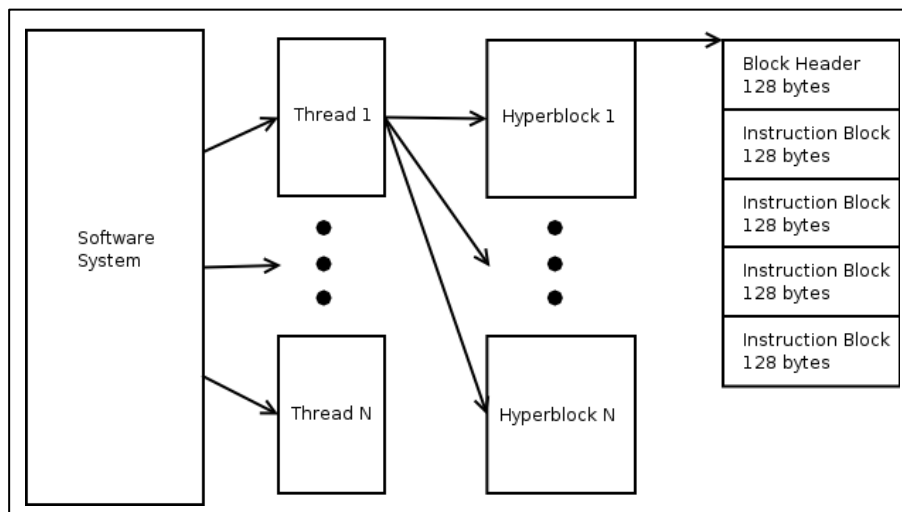


Figure 2: Partitioning of a threaded software system.

partitioning, the architecture allocates several physical frames to each thread; in the TRIPS prototype, threads evenly share the available frames (e.g. four threads use 16 frames each; two hyperblocks per thread). Within each thread, the frames may group into several A-frames to allow for speculation. Unlike row partitioning, however, frame-partitioned thread do not benefit from the high bandwidth from the instruction and data caches.

Executing multiple threads in parallel requires several copies of program context resources. Each thread requires a separate program counter, 128-register file, global shift history register, commit buffer, block control state, and return address stack. In addition, the register stitching logic and cache tags use thread IDs. Depending on how many parallel threads the system allows, the A-frame allocation logic requires augmentations (i.e. each thread holds some number of hyperblocks in its program and allocates into some number of A-frames in the core).

Thread Execution:

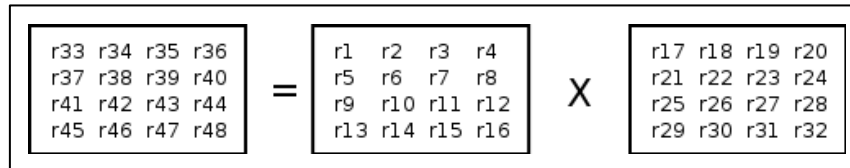


Figure 3: Overview of the matrix multiplication example. 'rXX' represents logical register XX out of the 128 logical registers.

This section gives an example to illustrate the TRIPS execution model for threads. Using the instruction set presented in [5, 7], a program multiplies two 4x4 matrices and stores the results in another 4x4 matrix. Each 4x4 matrix uses a set of registers (16 registers each) for storage as shown in Figure 3.

The first step in partitioning the matrix multiplier software is to identify the high-level instructions. Figure 4 shows the first high-level instruction; all other high-level instructions share the same synopsis as the first high-level instruction due to the high regularity of matrix multiplication. After identifying the high-level instructions, the next step is to break each high-

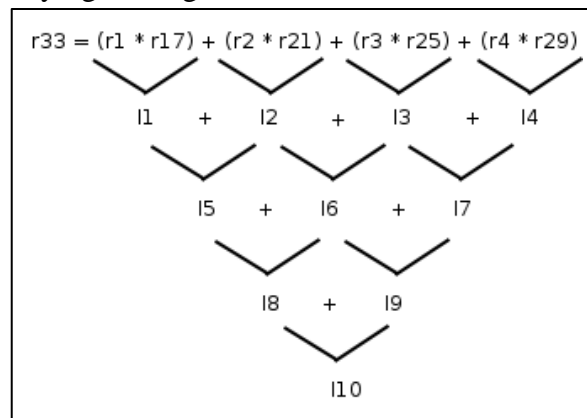


Figure 4: Instruction layout of the first matrix-multiply sequence.

level instruction into assembly instructions. Back to Figure 4, each high-level instruction consists of four multiply instructions and six add instructions, resulting in 10 assembly instructions per high-level instruction. With 16 matrix-multiply, high-level instructions, there are 160 assembly instructions in the software.

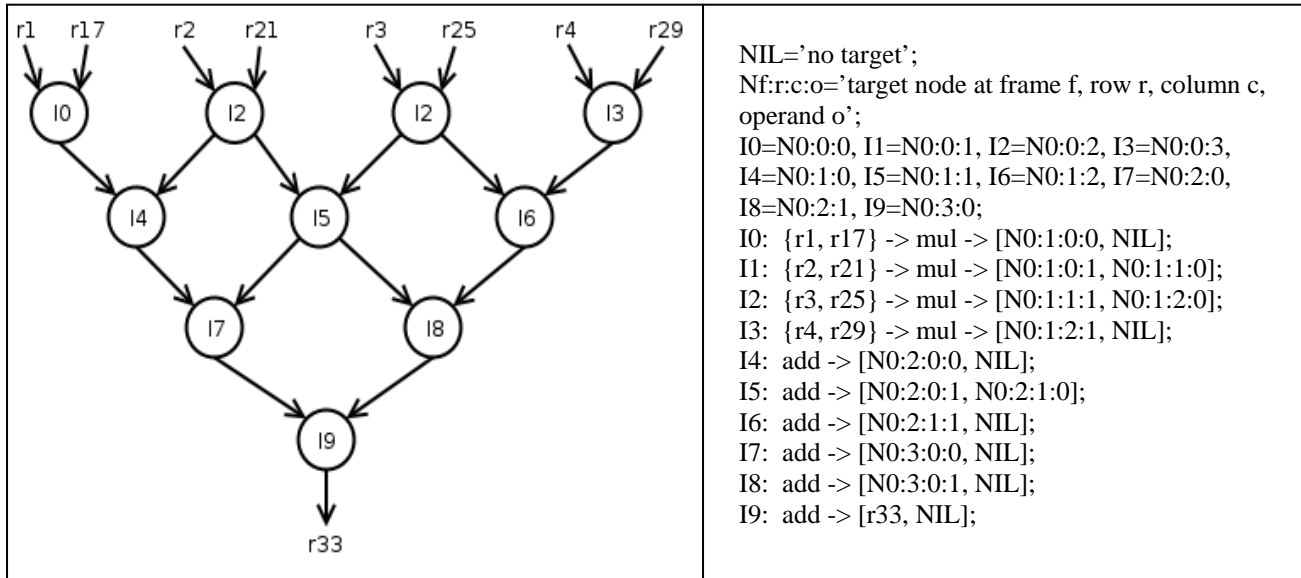


Figure 5: DFG of the first matrix multiply sequence (left). A naive grid node allocation and interconnection configuration for the first matrix multiply sequence (right).

The next partitioning step draws a dataflow graph (DFG) for each thread (see Figure 5). In DFGs, data ‘trickles’ down from the input, through the instruction nodes, and out as output data. Input data may start at any instruction node and output data may come from any instruction node. While many instruction nodes may share the same input data, each output (register) data must come from exactly one instruction node; however, it is possible for the same output data to come from several instruction nodes as long as all such instructions use predication or early block exits.

The DFG serves two purposes: it shows the data dependences between instructions, and it reveals the critical data path(s). In the case of the matrix multiplier, the critical data paths run from each of the multiply instructions down to the final add instruction. Critical data paths are important for evaluating certain metrics such as performance and reliability. For reliability, each output from each frame has a reliability factor that depends on the number of needed paths to the output and the length of each needed path; critical data paths have the worst reliability factors. A needed path is a path that is required to generate the correct output; furthermore, a needed output is an output required to maintain the correctness of the system. The next section covers more on reliability.

With each thread’s DFG, the threads partition into hyperblocks. In the TRIPS prototype, each hyperblock stores eight frames (16 instructions per frame). The matrix multiplier consists of 160 instructions and, based on the grid node allocation in Figure 5, executes 10 instructions per frame, utilizing 80 instructions per hyperblock (62.5% frame utilization for non-overlapping

high-level instructions). If the matrix multiplier is single-threaded, it partitions into two hyperblocks, each computing half of the 16 matrix multiplications. Both hyperblocks load into the grid array in separate A-frames, and the program counter points to the first hyperblock. After the first hyperblock commits its results to the first half of the register set in the result 4x4 matrix, the grid architecture de-allocates it, and the program counter points to the next hyperblock for execution. The matrix multiplier can easily partition into two threads with even frame sharing; although, the physical registers used will differ from the single-threaded case because each thread will have its own copy of the register file. Each thread may allocate 32 frames or four hyperblocks each on one core, and two program counters are active during operation. Another possibility is to run each thread on both processing cores, allocating eight hyperblocks to each thread and reducing contention for the ALUs.

Servers have copious number of threads running concurrently. While one thread may execute per core, TRIPS exploits TLP better with four threads executing per core (eight threads total in the prototype). In addition, the system may swap blocking threads with another thread loaded in the cores. Back to the matrix multiplier, it may divide into eight threads where each thread handles two matrix multiplications.

Evaluating Reliability:

This analysis only considers transient faults in the grid arrays. Any other part of the system that is faulty will cause the entire system to fail into an unrecoverable state (until the system sends a reset signal). Each node has two primary paths for data: an execution path and a hop path. A fault on the node's execution path will prevent the node from executing instructions and

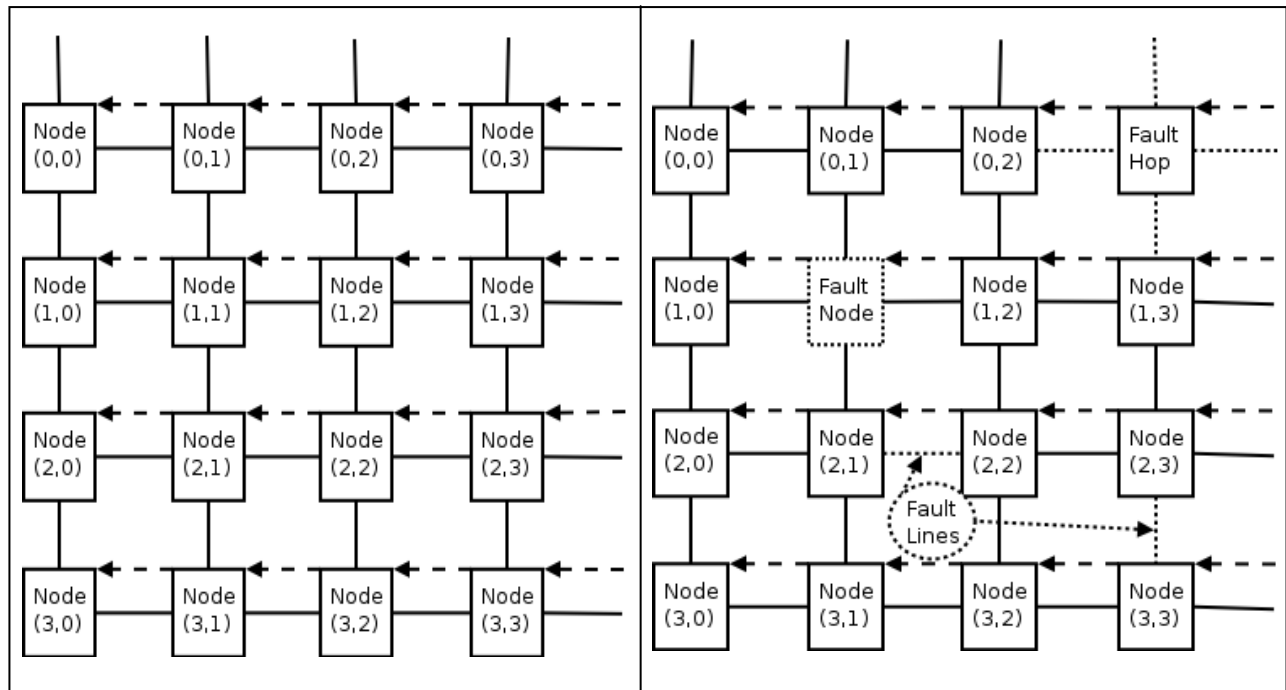


Figure 6: A fault-free grid array; register connections at topside, data cache connections at right side, instruction cache connections to each node (left). A grid array with a node with a faulty execution path, a node with a faulty hop path, and faulty inter-node connections (right). Notice that the inter-node connections surrounding the node with the faulty hop path are no longer available.

accepting/producing data. A fault in the node's hop path will prevent the node from bypassing data intended for other nodes while a fault on an inter-node connection only disables that connection. Figure 6 illustrates how certain faults affect the grid array. Finally, register bank connections route through the inter-node network via the nodes at the top row, data cache connections route through the same network via the nodes at right column, and the instruction cache connects directly to each node via a separate network. This paper assumes that each instruction cache connection is fault-free unless its target node's execution path is faulty.

To calculate the reliability of the grid array, Equation 1 factors in the reliability of each node and the inter-node network. The reliability of the grid array represents the probability that the circuit will execute any software correctly. Assuming there is no self-checking logic within or surrounding the grid array, every node and inter-node connection must be fault-free in order to assure correctness of the system. Conversely, if one node in the array is faulty, any instruction loaded in the faulty node will not execute correctly; therefore, the software depending on the instructions in the faulty node will fail. Likewise, the inter-node connections affect the software because all connections must be operational in order for each node to access the register banks, data cache memory, and operands from neighboring nodes. As a result, the reliability of the grid array drops significantly as its dimensions increase.

$$R_{Grid} = R_{Node}^{16} * R_{Interconnection}^{32} \quad (1)$$

While Equation 1 satisfies all cases of instruction placement in the grid, it also underestimates the reliability of the grid in certain cases. In other words, Equation 1 calculates the worst-case reliability of the grid array. Equations 2, 3, and 4 take a more fine-grained approach by calculating the reliability of each node with respect to the executing instruction. The reliability of each node has three aspects: the node's connection to the register banks, the node's connection to the data cache memory, and the node's paths to its operand target/source nodes. Depending on the instruction, a node may be accessing the register banks and/or the data memory, and the node may be targeting up to four remote nodes. There are many cases where not all reliability aspects factor into the node's overall reliability for the given instruction. For instance, a node executes an 'add' instruction, which does not require access to the data memory. Each node carries an execution reliability of R_{NExe} and a hop-path reliability of $R_{NHop/QHop}$ ($NHop$ refers to the hop path of a node while $QHop$ refers to the hop path plus the hop queue of a node).

$$R_{NReg,i,j} = R_{NExe} * R_{NHop/QHop}^i * R_{Interconnection}^{i+1} \quad (2)$$

$$R_{NData,i,j} = R_{NExe} * R_{NHop/QHop}^{(3-j)} * R_{Interconnection}^{(3-j)+1} \quad (3)$$

$$R_{NtoT,i,j} = R_{NExe}^{T+1} * R_{NHop/QHop}^{Nodes[\bigcup_{m=1}^T T_m]} * R_{Interconnection}^{Len[\bigcup_{m=1}^T T_m]} \quad (4)$$

Equations 2 and 3 represent the probability that the node can access the register banks and the data memory, respectively. The two equations make some assumptions about the nature of the connection between the node, the register banks, and the data cache. For both register banks and data cache, values from certain registers or memory may enter or leave the grid array at any row or column. For example, a node may access register X via the top row at columns 0, 1, 2, or 3.

The second assumption is that register and data memory accesses occur in a vertical or horizontal fashion, respectively (i.e. a node's register bank connection flows through the node's column and a node's data memory connection flows through the node's row); this is reasonable because there is no fault-detection logic in the array that can create 'detours' in the inter-node network. The conclusion drawn from Equations 2 and 3 is that nodes further away from the register banks and data memory are less reliable, taken that the instruction in execution requires register and/or data memory access.

Equation 4 calculates the probability that the node can reach all of its operand targets and sources. Due to the lack of hardware fault detection, the node's router will always take the same, shortest paths to the target/source nodes. T is the number of remote target/source nodes that the node connects to (whether the node is passing its result or hoping operands to the target/source nodes), T_m is the pre-fixed, shortest path between the node and target/source node m , $Nodes[X]$ is the number of nodes in path set X not counting the source and destination nodes, and $Len[X]$ is the number of interconnections in path set X . Equation 4 takes the union of all the paths to derive the total number of nodes and edges traversed between the node and its target/source nodes. In order for the node to be reliable, all target/source nodes must be operational and reachable via their pre-fixed, shortest paths. According to Equation 4, the closer the node is to its target/source nodes, the more reliable the node is.

The overall reliability of a node depends on how many nodes, hop paths, and inter-node connections a given instruction needs to execute. Equation 5 combines Equations 2, 3, and 4 by taking the union of the node's target/source node paths, path to the register banks ($Column_{i,j}$), and the path to the data memory ($Row_{i,j}$). $Column_{i,j}$ and $Row_{i,j}$ are empty path sets if the instruction does not access the register banks and the data memory, respectively; otherwise, $Column_{i,j}$ contains the path from the register banks down to row i along column j , and $Row_{i,j}$ contains the path from the data memory left to column j along row i .

$$R_{Node,i,j} = R_{NExe}^{T+1} * R_{NHop/QHop}^{Nodes[(\bigcup_{m=1}^T T_m) \cup Row_{i,j} \cup Column_{i,j}]} * R_{Interconnection}^{Len[(\bigcup_{m=1}^T T_m) \cup Row_{i,j} \cup Column_{i,j}]} \quad (5)$$

While Equation 5 applies to nodes executing an instruction but not hopping operands, Equations 6, 7, and 8 apply to nodes that only hop operands, execute an instruction and hop operands, and does not perform any operation, respectively. If a node is executing an instruction, its overall reliability must factor in its execution path; likewise for the node's hop path if the node is hoping operands.

$$R_{Node,i,j} = R_{NExe}^T * R_{NHop/QHop}^{Nodes[(\bigcup_{m=1}^T T_m) \cup Row_{i,j} \cup Column_{i,j}]} * R_{Interconnection}^{Len[(\bigcup_{m=1}^T T_m) \cup Row_{i,j} \cup Column_{i,j}]} \quad (5)$$

$$R_{Node,i,j} = R_{NExe}^{T+1} * R_{NHop/QHop}^{Nodes[(\bigcup_{m=1}^T T_m) \cup Row_{i,j} \cup Column_{i,j}]} * R_{Interconnection}^{Len[(\bigcup_{m=1}^T T_m) \cup Row_{i,j} \cup Column_{i,j}]} \quad (5)$$

$$R_{Node,i,j} = \emptyset \equiv Not_Applicable \quad (5)$$

For the next sections, Equations 5, 6, 7, and 8 combined will be the reliability model that measures each node's ability to be used for a certain frame. For a node to operate correctly, a transient fault cannot affect the behavior of the node must.

RX-Morph:

While keeping changes made to the TRIPS architecture minimum, RX-morph includes a polymorphous resource called a hop queue, or a hop buffer. The hop queue is a FCFS (First-Come, First-Serve) queue. Based on the reliability model, the hop queue brings data closer to its destination to increase the reliability of nodes with longer data paths.

Figure 7 shows a system level design of node with a hop queue. In the original node design, the hop path (the path from the node controller directly to the operand router) provided a route to form a combinational circuit among the nodes in the grid array; each combinational circuit executes per frame. As noted earlier in the reliability model, some software may lead to long data paths in the combinational circuit depending on what instructions execute and where the system places those instructions. With a hop queue en route of each node's hop path, the data path of each combinational circuit (each frame) shortens down to a single inter-node connection per path direction. Figure 8 illustrates how hop queues affect the data path of a register value and the result of an add instruction. At frame 0, the register value enters the first column and top row into the first hop queue. In the next frame, the register value traverses one node down to the next hop queue. The register value continues traversing in this fashion until it reaches the node with the add instruction. The add instruction processes the register value and passes its value to its consumer instruction using hop queues along the data path. If the hop queues were removed, the data path with respect to the node with the add instruction would be long, making the

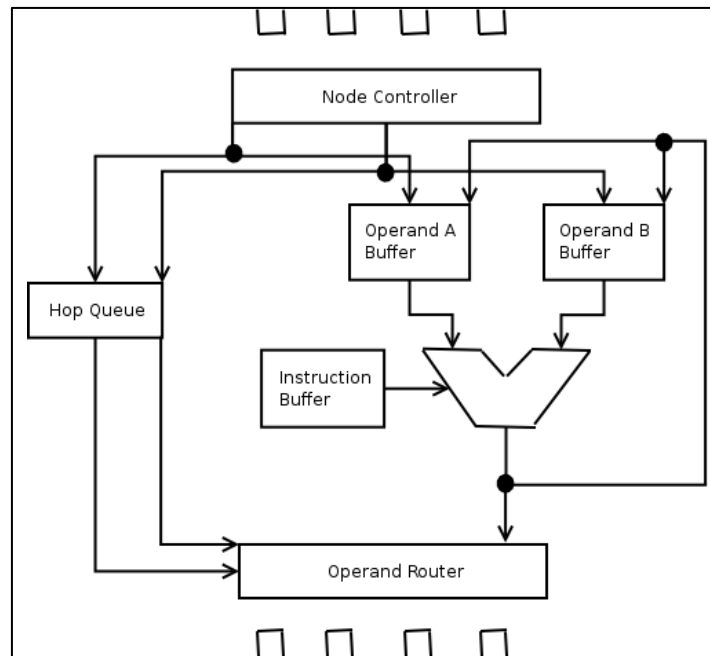


Figure 7: System-level design of a TRIPS node with a hop queue.

combinational circuit more prone to error. The next section compares the reliability of each node with and without the hop queues.

Although each hop queue holds each operand for one frame (the time it takes to execute a physical frame), the hop queues are transparent to the grid architecture. That is, no external logic is required to control the hop queues with the exception of enable signals; each hop queue's enable signal controls whether the hop path follows through the hop queue or bypasses it altogether, allowing the TRIPS architecture to configure between RX-morph and X-morph (non-reliable D-morph, T-morph, and S-morph). The reason behind the transparency is that nodes do not execute an instruction until the instruction's operands arrive. If the instruction's operands take several frames to arrive, then the execution of that instruction will delay for several frames. Consequently, the execution of the instruction's consumers will delay for several frames plus the number of nodes that the result of the producer instruction traverses.

When several instructions are operating, there may be more contention on the hop queues due to shared hop paths. It is possible for an operand to delay for more than one frame per node because the hop queues are FCFS. Also, having more instructions executing in the same frame may

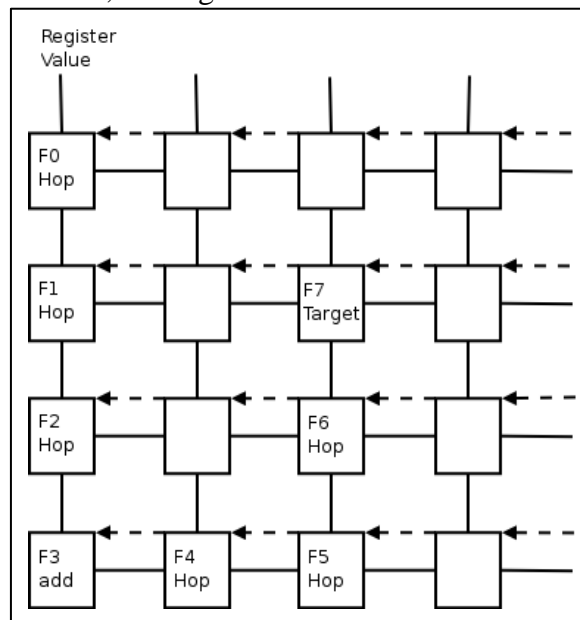


Figure 8: Data paths created by an add instruction positioned at node (3,0) with a target consumer at node (2,1). Four frames is needed to complete the add instruction and another four frames is needed to pass the result to the consumer.

require more inter-node connections and hop paths; while the reliability model describes the reliability of each node for an instruction, the reliability of the grid array for a frame may decrease because more components are in use. To improve the reliability of the grid array per frame, the compiler could schedule fewer instructions per frame in conjunction with enabling the hop queues. For example, a scheduling policy enforces a maximum of eight instructions per frame, utilizing half of the grid array in each frame.

Comparison Between RT-Morph and T-Morph:

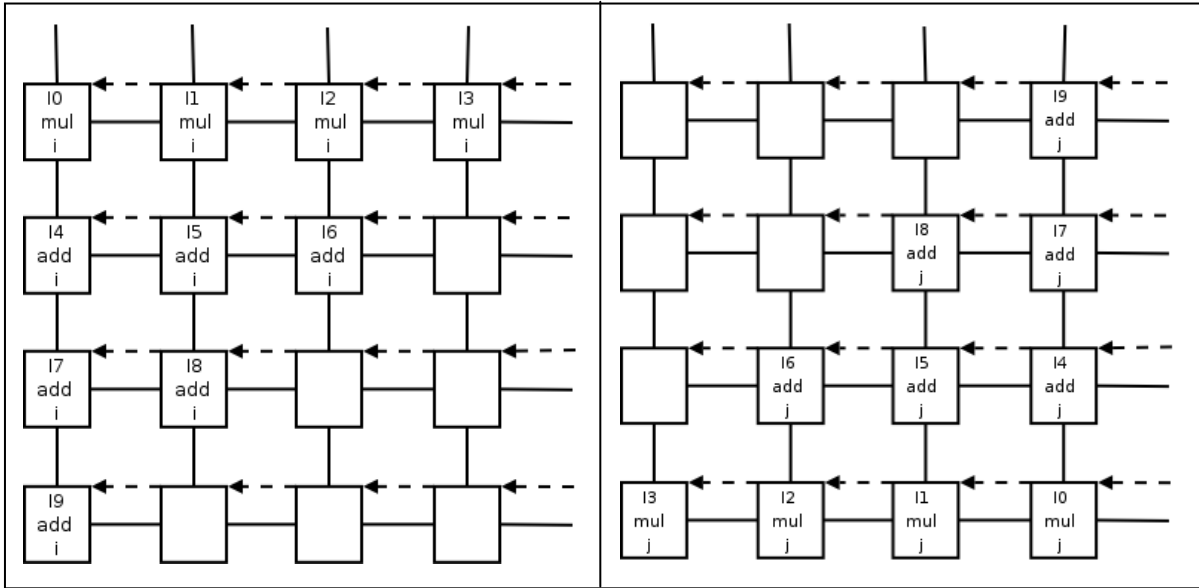


Figure 9: Two interleaving threads, i and j, occupy two frames. Thread i occupies the first frame (left). Thread j occupies the second frame (right). This pattern repeats for all interleaved threads.

This section revisits the matrix multiplier to calculate the average reliability of each node per frame using the T-morph and RT-morph. Figure 5 describes the instruction to node allocation used for each combinational circuit. The matrix multiplier is divided into eight threads, four threads per core and 20 assembly instructions (two matrix multiplications) per thread; each thread executes one hyperblock. For diversity in each core, two threads enforce the instruction to node allocation in Figure 5 while the other two threads invert the allocation horizontally. The

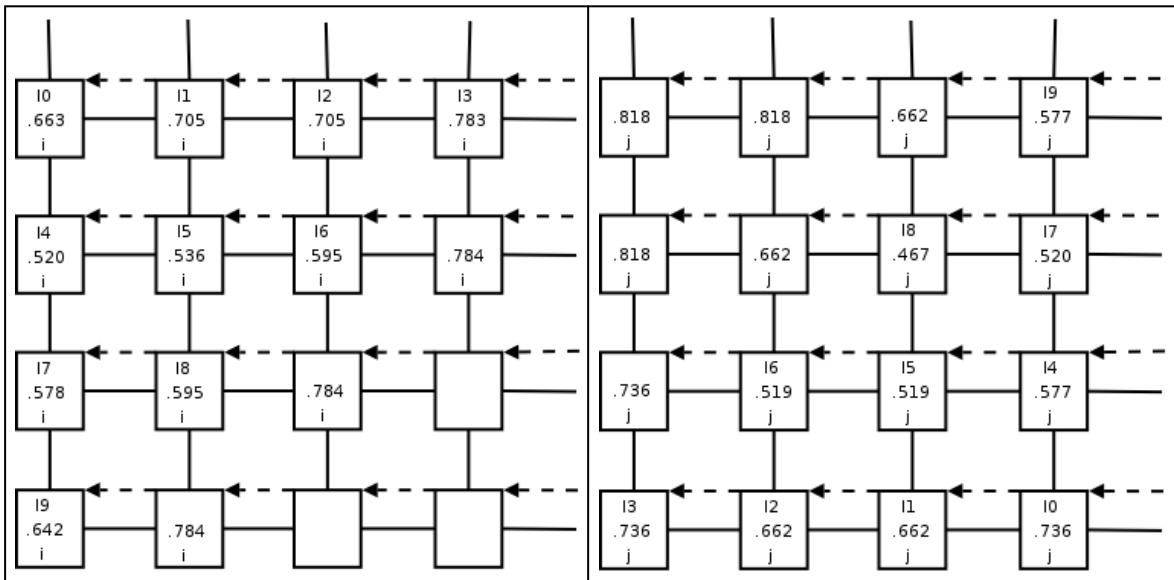


Figure 10: Based on Figure 9, the reliability of each node under the T-morph configuration. In each frame, $R_{NExe}=.900$, $R_{NHop}=.970$, and $R_{Interconnection}=.999$. The average node reliability for thread i (left) is .667. The average node reliability for thread j (right) is .656. Threads i and j combined yields an average node reliability of .661.

threads with inverted instruction to node allocation interleave during execution. Figure 9 shows how the instructions from two interleaving threads allocate to the nodes.

Figure 10 shows an example of how to calculate the average node reliability for T-morph. First, it calculates the reliability of each node, if the frame utilizes it, using the reliability model. Next, it derives the average node reliability for the frame. Because the pattern in Figure 10 repeats until all matrix multiplications complete, taking the average of the two interleaved threads calculates the overall average node reliability.

Before analyzing the average node reliability for RT-morph, this section accounts for the number of frames required. In T-morph, each matrix multiplication takes one frame to execute, resulting in eight frames per core to complete the full matrix multiplication. As for RT-morph, each matrix multiplication requires seven frames because the hop queues delay the hopping of operands; therefore, taking the average node reliability among the seven frames derives the average node reliability for a matrix multiplication. In addition, the total number of frames required by RT-morph is seven times greater than T-morph (56 frames per core for RT-morph).

Table 1 compares the average node reliability of T-morph and RT-morph by adjusting the reliability of the hop path used in T-morph (R_{NHop}) versus the reliability of the hop queue used in RT-morph (R_{QHop}). RT-morph is more reliable than T-morph when the reliability of the hop queue is zero to 15 percent less than the hop path if the reliability of the hop path is high (97 percent). When the reliability of the hop path is low (80 percent), the margin of reliability between the hop path and the hop queue increases to 25 percent.

Table 1: Average Node Reliability of T-morph and RT-morph

	$R_{NHop}=.97,$ $R_{QHop}=.97$	$R_{NHop}=.97,$ $R_{QHop}=.90$	$R_{NHop}=.97,$ $R_{QHop}=.80$	$R_{NHop}=.97,$ $R_{QHop}=.75$
T-Morph	.661	.661	.661	.661
RT-Morph	.876	.791	.677	.624
	$R_{NHop}=.90,$ $R_{QHop}=.90$	$R_{NHop}=.90,$ $R_{QHop}=.80$	$R_{NHop}=.90,$ $R_{QHop}=.70$	$R_{NHop}=.90,$ $R_{QHop}=.65$
T-Morph	.541	.541	.541	.541
RT-Morph	.791	.677	.624	.523
	$R_{NHop}=.80,$ $R_{QHop}=.80$	$R_{NHop}=.80,$ $R_{QHop}=.70$	$R_{NHop}=.80,$ $R_{QHop}=.60$	$R_{NHop}=.80,$ $R_{QHop}=.55$
T-Morph	.401	.401	.401	.401
RT-Morph	.677	.624	.523	.430

Note: $R_{NExe}=.90$ and $R_{Interconnection}=.999$

At the cost of decreasing performance by a factor of seven, RT-morph increases the average node reliability for T-morph by nearly 30 percent maximum. Some possible sources of improved reliability come from the assumption that only transient faults can occur. When a node utilizes shorter paths per frame, it exposes fewer components to faults. In Figure 9, for example, when a multiply instruction is processing and outputting its results under RT-morph, the multiply instruction is unaffected by transient faults occurring in the lower half of the grid array because

those faults may disappear in a later frame when the lower half is processing. In other words, each node is more resilient to transient faults by enforcing time redundancy.

Related Work:

The reliability model bases on the same factors as the optimized EDGE scheduler presented in [3]. Where the optimized EDGE scheduler enhances performance by locating instructions close to its sources and targets, the reliability model improve reliability by locating operands close to their destinations (i.e. consumer nodes, register banks, and data memory). An earlier work from the TRIPS architecture is the GPA-1 processor [2]. Each TRIPS core closely resembles GPA-1 except there are fewer inter-node connections and GPA-1 rearranges the circuit surrounding the grid array.

Another effort is the self-repairable inner product processor [1], which uses hardware partitioning to recursively design a more reliable multiplier. Lin constructs a 16 by 16 multiplier out of several 8 by 8 multipliers, which in turn are composed of 4 by 4 multipliers. Some other related works are the BioWall [6] and Sun's Niagara chip [4]. The BioWall is a self-healing grid of FPGAs where pressing onto the FPGAs' membranes injects faults. The Niagara chip is processor that is highly optimized for TLP using chip-multithreading (CMT).

Conclusion:

Using time redundancy and a polymorphous resource called the hop queue, RX-morph increases the average node reliability with the tradeoff of performance degradations. RX-morph is useful in harsh environments and when the reliability of each hop path is expected to be low; however, X-morph is more desirable for time-critical applications that can tolerant some degree of error (e.g. 3D games). For multithreaded servers, the difference between RT-morph and T-morph can mean the difference between maintenance and more maintenance.

References:

- [1] Lin, Rong and Martin Margala. "Novel design and verification of a 16 x 16-b self-repairable reconfigurable inner product processor." *ACM Press*, p. 172-7, 2002.
- [2] Nagarajan, Ramadass, et. al. "A Design Space Evaluation of Grid Processor Architectures." *IEEE Computer Society*, p. 40-51, 2001.
- [3] Nagarajan, Ramadass, et. al. "Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures," *pact*, pp. 74-84, 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04), 2004.
- [4] Nagarajayya, Nagendra. "Improving Application Efficiency Through Chip Multi-Threading." http://developers.sun.com/solaris/articles/chip_multi_thread.html.
- [5] Sankaralingam, Karthikeyan, et. al. "TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP." *ACM Press*, p. 62-93, 2004.
- [6] Tempesti, Gianluca, et. al. "The BioWall: an Electronic Tissue for Prototyping Bio-Inspired Systems." *IEEE Computer Society*, p. 221-30, 2002.
- [7] "TRIPS Website." <http://www.cs.utexas.edu/users/cart/trips/>.