Fault Aware Instruction Placement for Static Architectures

Premkishore Shivakumar   Divya P. Gulati   Calvin Lin   Stephen W. Keckler

Department of Computer Sciences
The University of Texas at Austin

Abstract

Aggressive technology scaling, rising clock frequencies, and the continued increase in microprocessor power density threaten both manufacturing yield rates and long-term reliability of integrated circuits. While defects in dynamically scheduled microprocessor architectures can be tolerated using mechanisms that are transparent to software, static architectures create different opportunities and challenges for reliability management. This paper proposes to expose the defective hardware configuration in a static architecture to the compiler, which can perform efficient fault reconfiguration through intelligent instruction scheduling. We conducted our studies on the TRIPS architecture whose computation core consists of a two-dimensional array of ALUs. The compiler generates blocks of instructions that are statically placed on the distributed ALU array, which are then executed dynamically in dataflow order. We consider two fault models, one in which only the computation elements can fail, and another that also allows faults in the communication channels. We examine scheduling algorithms that can avoid the faulty resources by exploiting the spatial redundancy inherent in the computation substrate. For a set of microbenchmarks, preliminary results demonstrate that our algorithms can reschedule the assembly code to tolerate execution unit faults with negligible loss in performance.

1   Introduction

Aggressive technology scaling, rising clock frequencies, and the continued increase in microprocessor power density threaten both the manufacturing yield rates and the long-term reliability of devices. Shrinking lithography, new materials and process technologies, and lower design tolerances increase the yield sensitivity to design features, and make integrated circuits more susceptible to manufacturing defects [7]. Further, rising core operating temperatures and power densities accelerate processor wear-out from intrinsic failure mechanisms such as electromigration, leading to reduced processor lifetimes [14].

Todays microprocessor based systems implement fault tolerance both at the system and the processor level. At the system level, they include hot spares for power supplies, processor chips, and memory modules, and rely on a service processor and the operating system for fault detection and reconfiguration. Recent microprocessor systems employ virtualization, and dynamic reconfiguration techniques such as dynamic CPU sparing to achieve goals of system performance, and availability [5, 4]. Fault tolerance at the processor level is typically implemented through purely hardware based techniques that are restricted to parity, ECC, and redundant rows in caches; and scrubbing and redundant bit-steering in the main storage. In the future, increasing demand for greater parallelism and faster clock rates will require microprocessors to distribute their resources and remove primitives that require single cycle global communication. We recognise that fault tolerance through purely hardware techniques in such processor architectures can lead to overheads in area, verification, and more importantly cycle time. First, we propose that future static architectures push dynamic fault reconfiguration within the boundaries of a single processor to achieve greater yield and system availability. Second, we propose that the defective processor configuration be exposed to the compiler which can then perform efficient fault reconfiguration by intelligent instruction placement. We argue that fault aware physical layout of the instructions can more effectively exploit the available spatial redundancy, and with fewer overheads than a purely hardware based approach to achieve both better performance and yield.

We conducted our studies on the TRIPS architecture whose computation core consists of a two-dimensional array of ALUs. The compiler generates hyperblocks [8] of instructions that are statically placed on the distributed ALU array which are then executed dynamically in dataflow order. We consider two fault models, one in which only the computation elements can fail, and another that also allows faults in the communication channels. We examine scheduling algorithms that can avoid the faulty resources by exploiting the spatial redundancy inherent in the computation substrate. For a set of microbenchmarks, preliminary results demonstrate that our algorithms can reschedule the assembly code to tolerate execution unit faults with negligible loss in performance. In this study, we focus only on defects exposed during the static compilation time, but we

see a natural path for extending the mechanism for dynamic compilation.

The remainder of this paper is organized as follows. Section 2 discusses how our work is related to, and extends other work in this area. Section 3 introduces the TRIPS architecture, discusses the fault model and describes the duties of a fault aware scheduling algorithm for this fault model. The base TRIPS scheduler algorithm, and our proposed fault aware algorithm are described in Section 4. We then briefly describe our methodology in Section 5. The experimental results are described in Section 6. Finally, Section 7 presents our conclusions.

2 Related Work

Yield management is typically done at multiple levels of the design. There are several ways in which additional *design for yield* guidelines can be incorporated to minimize the effects of common yield detractors at the layout and the circuit level [6]. Once these defects occur on the devices in the chip, techniques can be employed at the microarchitectural level to mask faults arising from these defects. In designs with a high degree of regularity such as DRAMs and SRAMs, it is common to make use of extra redundant rows to help improve yield. While defects in dynamic superscalar architectures can be tolerated using dynamic hardware mechanisms [12], static architectures provide an opportunity for software assisted yield management. In this paper, we propose to extend yield management in static architectures to the compiler level, thus enabling fault aware instruction placement and potentially achieving better yield and performance [12].

Dynamic reconfiguration on logically partitioned IBM pSeries symmetric multiprocessor systems allows movement of hardware resources from one partition to another enabling autonomic system management to optimize performance, resource utilization, and reliability [4]. It provides the foundation for self-healing and diagnosing software for dynamic CPU sparing that allows systems to transparently replace a defective processor with a fully functional processor with no impact on the application. The self-diagnosing software monitor the recoverable error rates for processors through firmware routines [4]. If the number of errors exceed an internal threshold, the operating system is notified which in turn triggers *dynamic reconfiguration* to substitute the defective processor.

We propose a more fine-grained technique where the error-detection mechanisms would first trigger a dynamic translation layer that attempts to isolate the fault within the processor. The dynamic translation layer we suggest is an instruction scheduler that can perform dynamic sparing of the defective processor resources, failing which the next layer or the operating system is triggered to replace the defective processor. The dynamic translation layer should be capable of extracting instruction blocks from the binary, and then remapping the block instructions to produce new assembly code and binary that will successfully execute on the faulty hardware by avoiding the defective resources. In this study, we focus only on defects exposed during the static compilation time and investigate scheduling algorithms that can perform the fault reconfiguration described above. Of course fail-in-place also requires techniques for detection and recovery from intermittent and transient failures that occur during a program's execution, and some such mechanisms are summarized in the literature [1, 10].

Traditionally, VLIW processors expose the processor pipeline details to the compiler, requiring all existing binaries to be recompiled following any change in the pipeline microarchitecture. The Transmeta Crusoe processor virtualized an X86 CPU by implementing a code morphing software layer that dynamically translated instructions from the target X86 ISA to the VLIW host ISA [5]. The Transmeta processor thus solved the problem by exposing the actual VLIW hardware configuration only to the code morphing software that does the dynamic translation. In general, dynamic translation can be used to isolate the details of the hardware from the software whenever the underlying hardware configuration is expected to change, thus enabling application portability to new environments and processors. In this paper, we extend this concept to reliability management in static architectures.

3 Yield Management in the TRIPS Architecture

We begin by describing the salient features of the TRIPS architecture. We then argue that the instruction scheduler is a suitable place for implementing fault tolerance in the TRIPS processor, and explore the duties of a fault aware scheduling algorithm for a specific fault model.

3.1 TRIPS Architecture

The TRIPS architecture [2] contains a two-dimensional array of computation elements connected by a thin mesh operand routing network. Each ALU includes an integer unit, a floating point unit, an operand router, and an instruction and operand buffer for storing instructions and their operands. The operand router follows dimension-order routing to communicate within the network. When routing a packet from a parent to a child node, the packet first travels in the x-direction (along the row) until it reaches the column of the child. Then it travels in the y-direction until it reaches the child node. Figure 1 illustrates the organization of the TRIPS core.

The TRIPS compiler generates hyperblocks [8] and schedules each hyperblock independently. An example schedule of instructions on a 2×2 execution array is shown in Figure 2. *Read* instructions are used to fetch values from the register file to the consumer instructions. Block register
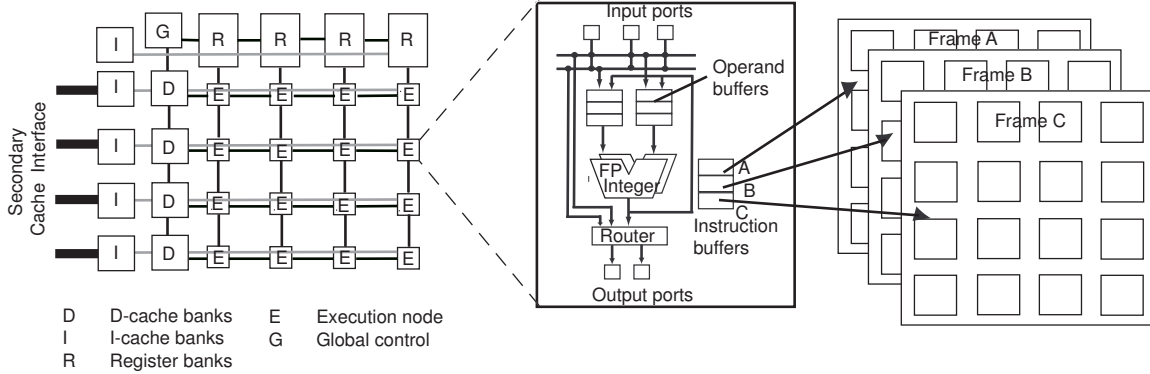
Figure 1. Example 4×4 TRIPS Processor.

D   D-cache banks   E   Execution node
I   I-cache banks   G   Global control
R   Register banks

outputs are produced by *write* instructions. In the TRIPS ISA, instructions do not encode their source operands, instead they explicitly encode the locations of their children. Figure 2 shows the instruction encoding for the above example. For example, the add instruction placed at location [0,1,0], upon execution, forwards its result to the LSH instruction placed at location [1,1,0].

The TRIPS architecture follows a dataflow execution model. The hardware fetches the actual instructions to the execution array, reads the input registers from the register file, and injects them into the appropriate ALUs. An instruction can fire once the instruction itself and all its operands have been received, which in turn forwards its results to consumer ALUs through the operand network on completion. Temporary values that are only live within a block are communicated directly from producer to consumer through the operand network; only register outputs are written to the register file.

Each ALU contains a fixed number of instruction buffer slots. We refer to corresponding slots across all ALUs collectively as a *frame*. A 4×4 grid with 64 instruction buffer entries at each ALU thus has 8 frames of 8 instructions each. A subset of contiguous frames constitutes an *architecture frame* (A-frame), into which the compiler schedules all instructions from a hyperblock. For example, dividing 64 frames into 8 A-frames composed of 8 physical frames each allows the scheduler to map a total of 128 instructions (per hyperblock) at once to the ALU array.

3.2   Fault Aware Instruction Placement

Error free execution can be achieved on a defective processor by forcing the program to utilize only the functional processor resources. Implementing a purely hardware based fault reconfiguration mechanism in partitioned static architectures, like the TRIPS architecture, can be quite inefficient. First, to handle failures in all of the statically scheduled resources a purely hardware based approach is required to replicate the fault isolation hardware in each of them. Further, a hardware based mechanism may be restricted by
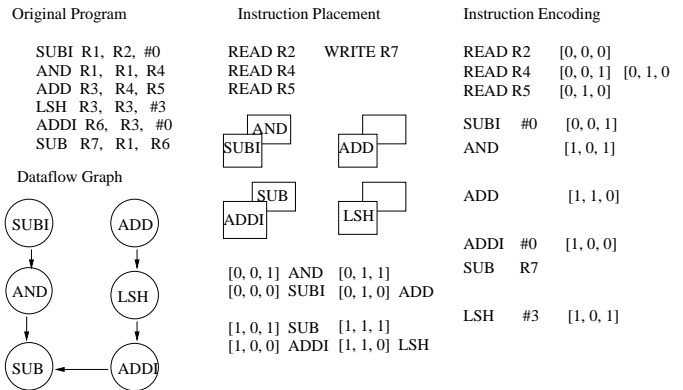

Figure 2. Instruction Physical Layout

area and complexity constraints to simple fault models, and solutions based on steering requests to explicitly provided spare resources, similar to caches.

On the other hand, the instruction scheduler in partitioned static architectures, like the TRIPS architecture, can examine the entire distributed execution substrate in evaluating all the constraints that contribute to optimal performance. The instruction scheduler can therefore naturally treat fault isolation as an additional constraint to the scheduling algorithm without any overheads in area, or dynamic execution time. The scheduler can also more effectively exploit the available redundancy and hence potentially scale better for complex fault models and greater number of defects. We argue that this visibility makes the instruction scheduler a natural target for implementing efficient fault reconfiguration.

3.3   Fault Model

In this study, our focus is only on the execution array in the TRIPS processor, one of the resources exposed to the compiler for static instruction placement. The execution array in the TRIPS processor occupies a substantial fraction of the processor area because each node contains a full set of integer and floating point functional units. While the layout and routing density of logic structures is less than regular
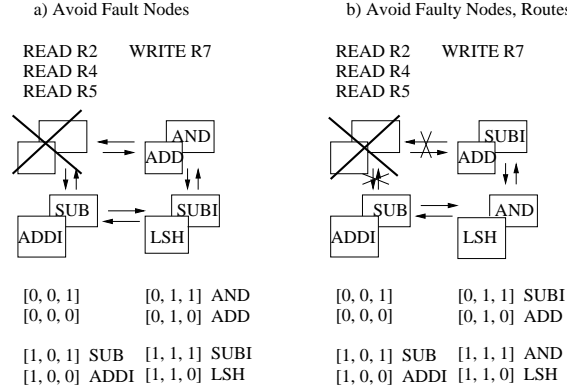
Figure 3. Fault Aware Instruction Placement

memory arrays, the large area occupied by the ALUs make it susceptible to defects. Further, the functional units are one of the hottest on-chip structures making them susceptible to intrinsic failures [14, 13].

The analysis in the rest of this paper assumes that an array contains at most one defective ALU in the execution array. Further, we consider two different granularities for a defect in the ALU node:

- Only the local functional units, the instruction buffer, or the operand buffers are defective.

- A more restrictive fault model which also allows defects in the operand router that transmits the results of computation to dependent instructions.

Depending on the defect granularity, a fault aware scheduling algorithm must not only avoid placing instructions on the faulty ALUs, but additionally ensure that no communication path between dependent instructions includes a node with a defective router. Figure 3 illustrates the two fault models, and the corresponding fault aware instruction placement for the same dataflow graph shown in Figure 2. While in the first case the algorithm only has to avoid placing instructions on the faulty ALUs, it can be observed in the second case (Figure 3.b) that the algorithm avoids the faulty communication paths also. For example, the `and` instruction is remapped to location [1,1,1], and the child `sub` instruction is remapped to [1,0,1] so that the communication path does not include the faulty node.

4 Fault Aware Instruction Placement

The base scheduler for the TRIPS architecture takes as input the instructions, and a detailed processor model that includes the routing topology, static instruction and communication latencies and produces the instruction schedule containing the assignment of instructions to ALUs. We begin by describing the base scheduler and then proceed to describe two simple heuristics for fault aware instruction placement.

4.1 Base Scheduler Algorithm

While a VLIW scheduler assigns each instruction an ALU and a time slot, the TRIPS scheduler assigns each instruction only an ALU without specifying a time slot. The base scheduler first computes the initial set of ready instructions, all of which can issue in parallel. These instructions are then ordered by their criticality, which is determined by their depth in the dataflow graph, and the instruction with the highest priority is selected for placement. The ALU assigned to the instruction is that which gives it the *earliest ready time* (ERT), which is calculated as follows:

$$ERT(i, alu) = \max_{\forall p}\{ECT(p) + Transmission\_Time[p, alu]\}$$

where p denotes a parent of i, $ERT(i, alu)$ is the earliest time at which the instruction can issue at this alu, $ECT(p)$ is the *earliest completion time* and refers to the expected time at which p will produce its results, and $Transmission\_Time[p, alu]$ denotes the time taken for the operand to reach instruction i at the node alu. To schedule i, the scheduler chooses the alu that minimizes ERT. After it schedules an instruction i, it adds to the ready set any of i's children whose parents have all been scheduled. It selects the next instruction for scheduling and iterates until completion. A detailed explanation of the base TRIPS scheduler is provided in [9].

4.2 Fault Aware Scheduler Algorithms

Fault aware scheduling algorithms rely on the redundancy in the execution substrate to isolate the fault. The nature of the fault model influences the redundancy needed to tolerate the potential failures. In the particular TRIPS processor configuration we consider, each hyperblock can contain a maximum of 128 instructions and can map upto 8 instructions on a single ALU. To tolerate one defective ALU in the execution array (Section 3.3), the algorithm needs to be able to remap the instructions scheduled on the defective ALU to other functional ALUs. Since the base scheduling algorithm may map upto 8 instructions on an ALU, the fault aware scheduling algorithm must find atleast 8 empty functional slots in the remaining nodes to provide the minimal redundancy for successful fault reconfiguration. This implies that each hyperblock can contain a maximum of 120 instructions (= 128 - 8) to have any redundancy at all for the algorithm to exploit.

The more restrictive fault model that allows both faulty ALUs and communication paths may prove this minimal redundancy to be inadequate even for mapping average sized hyperblocks on a TRIPS processor, for now each instruction has to not only find a functional ALU but also one that can be reached from all parents through functional routes. In this paper, we provide more than the minimal redundancy by compiling all the benchmarks to contain at most 112 instructions in each byperblock before the fault aware

scheduling algorithm is applied. This can potentially lead to performance loss even in the fault-free case and is the static cost of the technique, as we have to first create redundancy before it can be dynamically exploited when there are defects. As explained earlier, this is similar in concept to adding explicit spares for fault tolerance, which contribute to fixed overheads in area, and execution time. We now augment the base scheduler with two heuristics for the two fault models.

Avoid Faulty Nodes (AFN):   This heuristic assumes that only the ALUs can be defective, and everything else including the communication paths are operational. The scheduler takes as input a detailed processor configuration which additionally includes pointers to the defective ALU nodes, the rest of the inputs are identical to the base scheduler. The algorithm itself is identical to the base scheduler, but it now considers only the functional ALUs for instruction placement.

Avoid Faulty Links (AFL):   This heuristic accounts for both faulty ALUs and routing paths, and can be considered as an enhancement to the previous algorithm. The input processor configuration now not only contains pointers to the defective ALUs, but also has infinite communication latencies assigned to the defective nodes from all of its immediate neighbours. This implies that any path from a producer to a consumer instruction that includes a defective execution node is of infinite duration, which naturally serves to ensure that such a path is never selected to provide the instruction with the *earliest ready time* (ERT).

The constraints imposed by faulty communication paths and dimension-order routing can potentially lead to unsuccessful fault isolation frequently. Figure 4.a illustrates an example placement of the parent instructions and the faulty node, for which there is no node where the child can be placed that gives functional routes (that follow dimension-order routing) from both the parents. The base algorithm for instruction placement is identical to the base scheduler — but now some of the routes from parent instructions to children may be faulty. For every faulty path from a parent to the child instruction at this node, we insert an extra *MOV* instruction between the two. The *MOV* instruction is placed so that both the routes from the parent to it, and from it to the child node are fully functional. The algorithm presently chooses the first slot that satisfies the above condition, we recognize that more sophisticated choices are possible here. The only function of the *MOV* instruction is to pass the result from the parent to the child instruction and it now becomes the new child of the parent, and the new parent of the original child instruction. Figure 4.b illustrates one possible node for *MOV* insertion. The algorithm fails if it can find no node for successful *MOV* insertion.
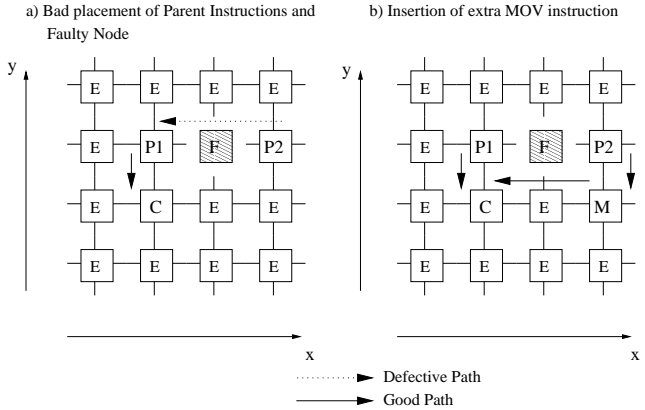
Figure 4. Figure 4a. shows that there is no path from the ALU corresponding to parent P2 to the child, since all paths have to pass through the faulty ALU. Figure 4b. shows the insertion of an extra MOV instruction that acts as an intermediate target. Both the routes from P2 to the MOV instruction and from the MOV instruction to the child are functional.

5   Methodology

The TRIPS processor [11] used in our study has an 4 x 4 array of execution nodes each with a full set of integer and floating point functional units. Each computation node also has 64 reservation stations, so that the overall array can accomodate eight blocks of 128 instructions each simultaneously. Further, a router resides on each of the 16 nodes to dynamically route results between dependent instructions taking 0.5 cycles per hop. The first level banked instruction and data caches are each 64KB, two-way, three cycle latency, and are located to the left of the execution array. We model a 13 cycle miss penalty to a 2MB L2 cache, and a 132 cycle main memory access time. The register file is also banked and is located at the top of the execution array.

Section 4 describes in detail both the baseline TRIPS instruction scheduler and the new fault aware scheduling algorithms. The applications we used for evaluating the scheduling algorithms are from the TRIPS microbenchmark suite [3]. Table 1 lists the microbenchmarks that we used in our evaluation. The microbenchmarks come from critical kernels in the SPEC 2000 suite, which are then compiled using the TRIPS compiler. It also includes hand optimized versions of some microbenchmarks that were formed by applying some transformations like loop unrolling more aggressively. In this preliminary exploration we are primarily interested in whether our fault aware scheduling algorithm is able to discover enough redundancy to remap the block instructions on the fully functional resources. To compare performance across the two schedules, we measure instructions per cycle (IPC) using a cycle accurate timing simulator that models the TRIPS architecture in detail.

Microbenchmark	Form
ammp 1	C source
ammp 2	C source
art 1	C source
art 2	C source
art 3	C source
art 1 hand	Hand optimized assembly code
art 2 hand	Hand optimized assembly code
bzip 1	C source
bzip 2	C source
bzip 3	C source
bzip 1 hand	Hand optimized assembly code
bzip 3 hand	Hand optimized assembly code
dhry	C source
equake	C source
equake hand	Hand optimized assembly code
gzip 1	C source
gzip 2	C source
matrix	C source
matrix hand	Hand optimized assembly code
parser	C source
sieve	C source
sieve hand	Hand optimized assembly code
twolf 1	C source
twolf 2	C source
twolf 1 hand	Hand optimized assembly code
twolf 2 hand	Hand optimized assembly code

Table 1. Microbenchmark Suite

Micro Benchmarks	Original AFL Heurisitic			Modified AFL Heursitic
	Total MOV Inst/Program	Max. MOV Inst/Block	Avg. MOV Inst/Block	Total MOV Inst/Program
ammp 1	0	0	0	0
ammp 2	0	0	0	0
art 1	0	0	0	0
art 2	2	2	0.17	2
art 3	0	0	0	0
art 1 hand	20	16	1.11	14
art 2 hand	0	0	0	0
bzip 1	0	0	0	0
bzip 2	0	0	0	0
bzip 3	0	0	0	0
bzip 1 hand	2	2	0.14	1
bzip 3 hand	0	0	0	0
dhry	0	0	0	0
equake	4	2	0.18	4
equake hand	3	3	0.18	3
gzip 1	0	0	0	0
gzip 2	0	0	0	0
matrix	0	0	0	0
matrix hand	1	1	0.05	1
parser	1	1	0.08	1
sieve	0	0	0	0
sieve hand	10	10	0.71	6
twolf 1	4	3	0.21	4
twolf 2	1	1	0.11	1
twolf 1 hand	8	6	1.14	7
twolf 2 hand	2	2	0.14	2

**Table 2. Extra MOV Instructions**

6 Preliminary Results

The *Avoid Faulty Nodes* algorithm succeeds in finding a legal schedule for all the microbenchmarks. The hyperblocks in all the microbenchmarks contain at most 112 instructions and hence can be mapped to fit on 14 ALUs each with 8 instructions. Since there is only one defective node (and no faulty links) there is always enough redundancy to remap the instructions.

The *Avoid Faulty Links* heuristic succeeded for 24 out of 25 microbenchmarks. Table 2 shows the extra *MOV* instructions inserted in the microbenchmark schedule to route around the defective nodes and links. The total number of *MOV* instructions inserted in all the microbenchmarks varies between 0 and 20; the maximum number of *MOV* instructions in any single block varies between 0 and 16. For instance, the heuristic inserts 16 extra *MOV* instructions in a single hyperblock of art_1_hand to successfully remap the instructions. Table 2 also shows that less than two *MOV* instructions are inserted in each hyperblock on the average, with most benchmarks successfully rescheduled without inserting any extra instructions at all. This demonstrates that, for this specific fault model, restricting each hyperblock to 112 instructions provides ample redundancy in most cases for effective intra-processor fault reconfiguration. The heuristic succeeds in scheduling the hyperblocks in the microbenchmarks with greater than 90 instructions, showing promise of scaling to real benchmarks.

The *Avoid Faulty Links* heuristic fails to schedule a hyperblock in the hand optimized assembly benchmark $sievehand$ that has 102 instructions in it. The algorithm failed after it had already added ten *MOV* instructions, and was unable to find a suitable node to map another. We modified the *Avoid Faulty Links* heuristic slightly so that when there is no node for the child that gives functional routes from all its parents, the child node that has the least number of defective routes, and hence needs the least number of *MOV* instructions, is chosen. Although the total number of extra *MOV* instructions decreases, as shown in the last column of Table 2 (Modified AFL Heuristic), the heuristic still fails to remap $sievehand$.

Since both the variants of the algorithm consider exactly one instruction at a time for scheduling, they can potentially arrive at a bad intermediate schedule where there is neither a node for the child instruction that provides functional paths between the child and all its parents, nor is there a suitable node for *MOV* insertion between the parent-child pair with the faulty path. Making the algorithm more sophisticated without adversely affecting the scheduling latency, or adding lookahead to the scheduling algorithm will increase its robustness but still cannot ensure its success. Figure 5 shows a way for ensuring a successful schedule for every hyperblock. By avoiding scheduling instructions on all the

Avoid Row and Column of Faulty Node
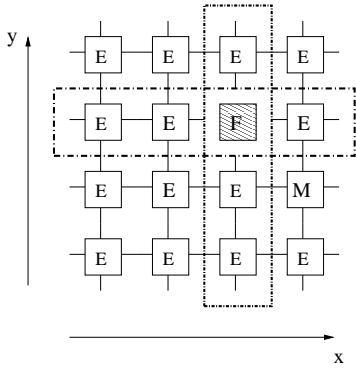
Figure 5. Avoid all the nodes in the same row or column as the faulty node during scheduling. This ensures that no communication path between a parent and child can include the defective node.

nodes in the same row or column as the faulty node we can ensure that no two dependent instructions include the defective node in their communication path. For this to be possible, each hyperblock can now contain a maximum of 72 (= 128 - 7x8) instructions. Although this is simple, it is both overly conservative and can potentially have a large negative impact on performance.

In general, we have only investigated a simple fault aware scheduling algorithm that reschedules all the instructions in every block avoiding all faulty execution nodes, and communication paths between dependent instructions that include faulty nodes. This is only one solution in the spectrum of possible scheduling algorithms that optimize concurrently for performance, availability, and scheduling latency. Algorithms that reschedule the entire block of instructions can potentially offer higher performance and yield, but may have considerable scheduling latency. At the other extreme, algorithms that aim to remap the minimum number of instructions will incur smaller scheduling latencies but will likely have poorer yield and performance.

Both the AFN and AFL heuristics show a very slight (approximately 1%) drop in performance compared to the base scheduler with no faults.

7 Conclusion

This paper proposes to enhance yield and enable graceful degradation of fail-in-place systems through efficient compiler assisted fault reconfiguration in future microprocessors. We discuss the trade-offs between scheduling latency, performance and system availability, and evaluate a simple scheduling algorithm that remaps all the instructions to avoid the faulty resources but optimizing only for performance. Our preliminary results demonstrate, on a set of microbenchmarks, that we are able to remap the assembly code to avoid the defective execution units and paths on a

TRIPS processor with atmost one defective node with negligible loss in performance.

Our compiler assisted solution for exploiting redundancy and enabling fault aware instruction placement is synergistic with many proposed design ideas for performance, low power, and reliability. Future wire delay dominated architectures may be required to use the compiler in spatially scheduling the instructions on the distributed execution substrate to achieve scalable performance by explicitly accounting for the communication latencies between dependent instructions [11, 15]. The Transmeta Crusoe processors judiciously traded performance for low power consumption by an innovative partitioning of the microprocessor functions between software and hardware [5]. Finally, dynamic reconfiguration features have provided the foundation for the relatively coarse grained self-healing and diagnosing features built into the IBM pseries 690 servers in AIX 5.2 [4].

We are planning to extend this preliminary investigation in several ways. A detailed area, yield, and lifetime reliability model will be required to determine more precisely the fault model that primarily influences the complexity of the scheduling algorithm. Our preliminary experiments use microbenchmarks, and it will be interesting to investigate how the nature of applications influence the complexity of the scheduling algorithm. We plan to extend our evaluation to encompass the floating point and integer SPEC benchmark suite. We view these as interesting opportunities for future work in this area.

Acknowledgments

References

[1] AUSTIN, T. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. *International Symposium on Microarchitecture* (November 1999), 196–207.

[2] BURGER, D., KECKLER, S., MCKINLEY, K., DAHLIN, M., JOHN, L., LIN, C., MOORE, C., BURRILL, J., MCDONALD, R., W.YODER, AND THE TRIPS TEAM. Scaling to the End of Silicon with EDGE Architectures. p. 37:7.

[3] CHEN, X. Trips Microbenchmark suite.

[4] JANN, J., BROWNING, L. M., AND BURUGULA, R. S. Dynamic Reconfiguration: Basic Building Block for Autonomic Computing on IBM pSeries Servers. *IBM Systems Journal, Vol 42, No 1* (March 2003), 29–37.

[5] KLAIBER, A. The Technology Behind Crusoe Processors. *Transmeta White Paper* (January 2000).

[6] KOREN, I., AND KOREN, Z. Defect tolerant VLSI circuits: Techniques and yield analysis. In *Proceedings of the IEEE* (September 1998), vol. 86, pp. 1817–1836.

[7] LI, X., STROJWAS, A. J., AND ANTONELLI, M. F. Holistic Yield Improvement Methodology. *Semiconductor Fabtech Journal 8*, 7 (July 1998), 257–265.

[8] MAHLKE, S., LIN, D., CHEN, W., HANK, R., AND BRINGMANN, R. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture* (June 1992), pp. 45–54.

[9] NAGARAJAN, R., KUSHWAHA, S. K., BURGER, D., MCKINLEY, K. S., LIN, C., AND KECKLER, S. W. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques* (October 2004).

[10] REINHARDT, S. K., AND MUKHERJEE, S. Transient Fault Detection via Simultaneous Multithreading. In *International Symposium on Computer Architecture* (July 2000), pp. 25–36.

[11] SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S., AND MOORE, C. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (June 2003), pp. 422–433.

[12] SHIVAKUMAR, P., KECKLER, S. W., MOORE, C. R., AND BURGER, D. Exploiting microarchitectural redundancy for defect tolerance. In *The 21st International Conference on Computer Design* (October 2003).

[13] SRINIVASAN, J., AND ADVE, S. V. Predictive Dynamic Thermal Management for Multimedia Applications. *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS 2003)* (June 2003).

[14] SRINIVASAN, J., ADVE, S. V., BOSE, P., AND RIVERS, J. A. The Case for Microarchitectural Awareness of Lifetime Reliability. *In Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2004).

[15] WAINGOLD, E., TAYLOR, M., SRIKRISHNA, D., SARKAR, V., LEE, W., LEE, V., KIM, J., FRANK, M., FINCH, P., BARUA, R., BABB, J., AMARSINGHE, S., AND AGARWAL, A. Baring It All to Software: RAW Machines. *IEEE Computer* (September 1997), 86–93.