The Effects of Explicitly Parallel Mechanisms on the Multi-ALU Processor Cluster Pipeline

Andrew Chang, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, Whay S. Lee†

Computer Systems Laboratory
Stanford University
Gates CS Building
Stanford, CA 94305

†Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

{achang, billd, skeckler, npcarter, wslee}@cva.stanford.edu

Abstract

Continuing reductions in on-chip geometries yield increasing numbers of transistors per chip and fundamentally faster devices but also result in effectively slower wires. This combination presents significant challenges for new microprocessor architectures. The disparity in performance between on-chip arithmetic units and memory creates longer effectively latencies. The changing balance between gate delay and wire delay penalizes global interactions. The MIT Multi-ALU Processor (MAP) architecture incorporates three explicitly parallel mechanisms to address these challenges. Efficient intercluster interactions enable instruction scheduling across clustered arithmetic units. Deferred exceptions based on ERRVAL's facilitate aggressive instruction reordering and speculation. Zero-cycle multithreading provides latency tolerance without sacrificing single threaded performance. In this paper, we describe each of these mechanisms and quantify their impact on the area and routing of the cluster pipeline in the 5 Million transistor MAP chip. Zero-cycle multithreading accounts for over 44% of the total cluster area. Support for ERRVAL's requires very little area (less than 4%). The intercluster interaction mechanisms require minimal cluster area and less than 5% of the available global routing resources, but enable fully general access across clusters and between all arithmetic units.

1 Introduction

As interconnect delay becomes an increasingly important limit to processor performance, new innovations are needed in microprocessor architectures to minimize global interactions, enable explicit instruction scheduling, and tolerate wire latencies. One approach, compiler-scheduled explicit parallelism, simplifies instruction issue and enables distributed decision logic. While, compiler based instruction scheduling has been used in VLIW supercomputers [3], and has been proposed for new architectures such as the Intel EPIC [7], additional mechanisms are needed for effective scheduling across distributed clusters of arithmetic units. Compilers can aggressively uncover more Instruction Level Parallelism (ILP) by reordering instructions above or below branches [11], and by scheduling instructions speculatively. However, unnecessarily triggered exceptions can negate the performance benefits. Multithreading can provide latency tolerance and improve utilization by interleaving the execution of multiple instruction streams [6, 1]. It can also improve performance by simultaneously combining instruction and thread level parallelism [8, 10]. Unfortunately, many existing implementations of multithreading penalize single-threaded performance.

The MIT Multi-ALU Processor (MAP) incorporates three explicitly parallel mechanisms to address these challenges. The MAP's distributed execution units and register files efficiently communicate and synchronize through a small set of *intercluster interactions*. The introduction of a tagged error-value (*ERRVAL*) datatype simplifies exception deferral. *Zero-cycle* multithreading combines instantaneous thread switches with cycle-by-cycle dynamic thread selection without sacrificing single-threaded performance. While the performance benefits of these features have been detailed separately [9, 8], this paper describes each mechanism and quantifies the associated logic area costs and routing requirements. Section 2 provides a brief overview of the organization of the MAP chip, including the partition of global and local resources and the composition of each cluster. Section 3 details the three mechanisms and their usage. Section 4 describes the cluster pipeline. Section 5 evaluates the costs associated with implementing each mechanism in the MAP chip. Finally, Section 6 summarizes the three key mechanisms and discusses their applicability to mainstream microprocessor pipelines.

2 Description of MIT Multi-ALU Processor

The MAP maximizes on-chip performance by simultaneously supporting parallelism at all granularities. As shown in Figure 1, the MAP[1] contains three independent

[1]The original MIT MAP architecture [4] included four symmetric processing clusters; a quad-banked memory system; the two switches; and a network interface/3D-mesh-router.

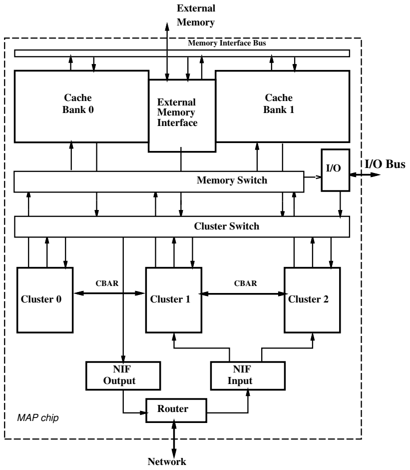

Figure 1. Block Diagram of the MAP

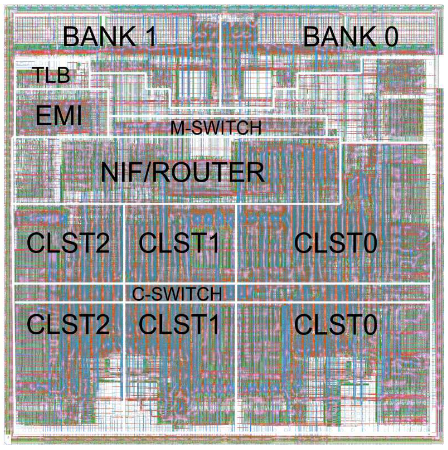

Figure 2. Plot of the MAP Chip

processor clusters, each with three execution units; a dual banked 32-KByte unified cache; two global communication switches (Cluster Switch and Memory Switch); a network interface/2D-mesh-router; and an I/O controller. The clusters can directly write into each other's register files and receive results from memory system and I/O requests through the Cluster Switch. They submit memory and I/O requests using the Memory Switch. The clusters can perform global synchronization through a single `cbar` instruction enabled by a set of global CBAR wires.

Figure 2 shows a plot of the actual 5 Million transistor MAP chip which measures 18.25mm $\times$ 18.30mm. Due to on-chip area constraints, the clusters in the actual chip are asymmetric. Cluster 0 is complete and contains three execution units - integer (IU), memory (MU), and floating-point (FPU), while clusters 1 and 2 lack the floating point-pipelines. The MAP chip was released for manufacturing in June 1998 and is being fabricated in a 0.7μm drawn, 0.5μm effective, 5-level metal, CMOS process. The MAP chip is the core processing element for each node of the planned 16-node prototype M-Machine multicomputer.

Cluster Organization: As illustrated in Figure 3, Cluster 0 is equivalent to a basic 64-bit microprocessor without a data cache. The cluster consists of three local arithmetic units, three register files, and a 4-KByte instruction cache. The IU contains a 64-bit ALU and a barrel shifter while the MU contains a 64-bit ALU and a load/store unit. The FPU consists of a fully-pipelined, four-cycle multiplier-accumulate unit and a non-pipelined iterative divide/square-root unit. Both the Integer and Floating-Point units can write results to remote clusters via a shared Cluster Switch port. The MU and the Instruction Cache submit fetch requests through a shared Memory Switch port. Each instruction contains up to three operations (one per arithmetic unit) which are issued simultaneously and in lock step. While, issued in order, instructions may complete out of order due to differences in execution latencies.

The data within a cluster is stored in three register files. The integer register file (IRF) contains 5 banks of 16 registers[2] and can be read and written independently by both the integer and memory units. The floating-point register file (FRF) also includes five banks of 16 registers. The floating-point unit can read three operands (for multiply-accumulate) and write one result to the FRF. The memory unit can read one operand from the FRF for floating-point store operations. The local Cluster Switch port can write into either the IRF or the FRF. Each cluster also has five banks of 16 one-bit condition code (CC) registers. These registers hold the results from comparison operations and are used for conditional branches and predication.

Concurrency is exploited within a cluster using multi-threading. In each cluster pipeline, up to five threads may be simultaneously loaded into dedicated *thread slots* consisting of private pipeline registers and unique sets of integer, floating point and CC register banks. Instructions from different threads are dynamically interleaved over the cluster's execution units. A total of 15 thread slots are implemented across the three MAP clusters, and an instruction stream may be installed in any thread slot. The thread slots on separate clusters are organized into five *thread groups* consisting of one thread slot from each cluster. The intercluster interactions discussed in this paper are only applicable *within* a single thread group.

Global Interconnections: The Cluster Switch and the Memory Switch provide the communication paths between the MAP's processing units and storage modules. The Cluster Switch, shown in Figure 1, is a crossbar composed of three global busses, each writing into a different specific

[2]In the IRF, i0 is hard-wired to Zero and i1 is set to the value of the program counter. Similarly, f0 is hard-wired to Zero in the FRF.

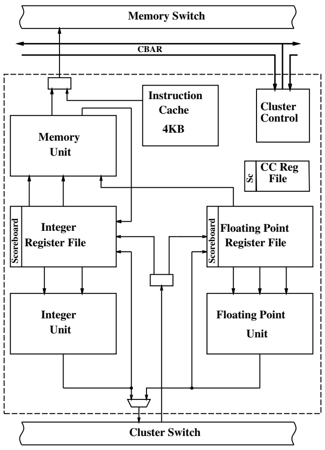

Figure 3. Block Diagram of Cluster 0

cluster. A cluster uses the switch to write data directly into another cluster's register file and to transfer data between its own integer and floating-point register files. The memory banks, the external memory interface, and the I/O unit use the Cluster Switch to return data to the clusters in response to prior requests. All seven data sources arbitrate for the Cluster Switch one cycle prior to sending the data, and loss of arbitration causes the source unit to stall. The Memory Switch is a crossbar implemented with two busses. Each bus writes into one of the on-chip cache banks. The clusters transmit load and store requests to the memory system via the Memory Switch after successfully arbitrating for the bus connected to the bank indicated by the address. Both arbitration and data transfer on the Memory Switch can occur in the same cycle. The combination of the Cluster and Memory Switches allows multiple clusters to access multiple cache banks simultaneously with a minimum load-use latency of three cycles, including both switch traversals.

Clusters can perform fast synchronization through a separate global CBAR mechanism. Threads on different clusters but within the same thread group can initiate a barrier by issuing a cluster barrier (`cbar`) instruction. A set of dedicated CBAR wires transmit the synchronization condition to all three clusters. Each thread group has its own unique set of CBAR wires. As a result, the MAP chip can support simultaneous barriers in multiple thread groups and allow an arbitrary mix of synchronizing and non-synchronizing thread groups.

3 Explicitly Parallel Mechanisms

The MAP architecture incorporates three mechanisms to enable efficient compiler-driven instruction scheduling. Intercluster interactions enable fast explicit communication and synchronization between decoupled clusters of execution units. ERRVAL based deferred exceptions provide a simple means to preserve fault state and enable aggressive instruction reordering and speculation by eliminating unnecessary handling penalties. Zero-cycle multithreading provides latency tolerance and allows instantaneous thread-switch while preserving single-threaded performance.

Intercluster Interactions: Four features collectively enable efficient intercluster interactions: local register scoreboarding, remote-register write capability, the `empty` instruction, and the `cbar` instruction. The combination of the first three enables efficient explicit communication, while `cbar` allows fast intercluster synchronization. The local register scoreboard unifies the management of both local and remote register writes. A register is present (full), if it has been updated and is absent (empty) if it is awaiting the update. The empty state of a register will stall any instruction which attempts to read that register before it is updated. *All* integer and floating-point arithmetic instructions can specify either a local or remote register as their destination. A local operation, upon issue, implicitly marks the destination register as empty and, upon completion, automatically sets the presence bit to full. For remote-register writes, the compiler must use the `empty` instruction to explicitly empty the destination register in the remote cluster. The source cluster can not implicitly invalidate this register as it only has local instruction schedule information. It would need a global interaction path to the destination cluster to eliminate a potential read-after-write hazard. When the remote-write completes, the local Cluster Switch port at the destination marks the register full and any stalled instruction stream may then proceed.

In both superscalar and VLIW processors, barriers are implicit. Ordering is guaranteed by hardware in the former and by the lock-step issue policy of the latter. However, in the MAP chip, lock-step issue is only maintained within a cluster. The `cbar` instruction provides an explicit barrier. The insertion of a `cbar` into the instruction stream in each of the three clusters implements a fast barrier among threads within a thread-group. When a `cbar` reaches the issue point in a cluster, it stalls that thread until the associated threads in each of the other clusters have also reached their own `cbar` operations. Each cluster then issues its respective `cbar` with no side-effects and proceeds normally. Any instruction scheduled after the `cbar`'s is guaranteed to issue only after the barrier has occurred. The clusters signal each other using the CBAR wires. At the cost of a single issue slot and the CBAR wires, this mechanism guarantees the ordering of instructions for threads within a thread group, across clusters, and independent of dynamic execution effects.

ERRVAL Enabled Deferred Exceptions: The MAP architecture uses a segmentation-based memory system and implements unforgeable *guarded pointers* for fast capability-based addressing [2]. The ERRVAL is a specific type of

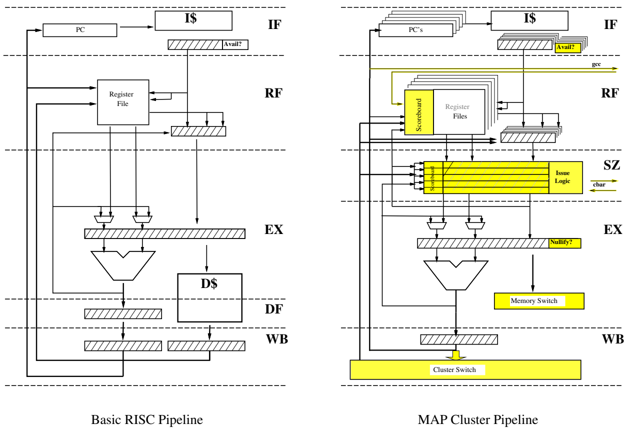

Figure 4. Basic RISC vs. MAP Cluster Pipeline

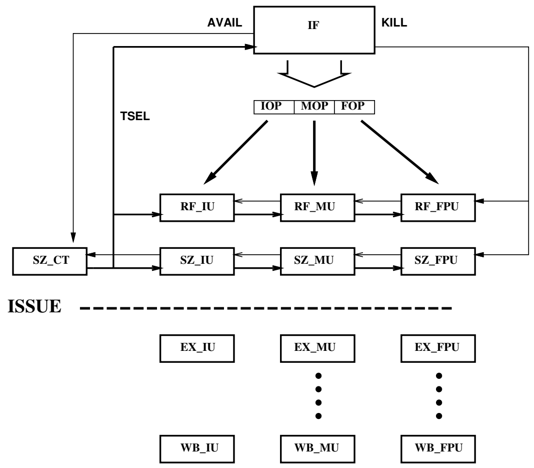

Figure 5. Distributed Implementation

guarded pointer and, for several common protection and segmentation violations, encodes a distinct error code; the identifier of the offending thread; and the program counter value. In each faulting case, hardware detects the error condition and automatically creates the appropriate ERRVAL. Instead of triggering an exception, the ERRVAL is written back as the result for the faulting instruction.

ERRVAL's are allowed to freely migrate throughout the MAP architecture. As with a regular data type, they can be loaded and stored into registers/memory and off-chip I/O, or be transmitted in messages between MAP nodes. All arithmetic operations can take ERRVAL's as operands. If one operand is an ERRVAL, the operation simply propagates that ERRVAL as the result. If an operation has multiple ERRVAL operands, it arbitrarily passes one of them.

An ERRVAL only triggers an actual exception when continued propagation is impossible, for example: stores to an ERRVAL address, comparison operations with one or both ERRVAL operands, and final resolution of branch/jump operations. If desired, the occurrence of ERRVAL's can be easily detected using a provided `iserr` instruction allowing programmers to test the results of registers to detect ERRVAL's and control their propagation.

Zero Cycle Multithreading: The MAP employs zero-cycle multithreading to enhance performance. Each cluster accommodates up to five independent threads simultaneously. Each of these can be switched into execution instantaneously, allowing efficient overlap of computation onto latencies of even a few cycles. Every cycle, zero-cycle multithreading dynamically adapts the selection for instruction issue to the number of ready and available threads. No issue slots are granted to stalled threads. When only a single thread is installed, zero-cycle multithreading maximizes performance by allowing that thread to issue instructions whenever its operands are present and the execution resources are available. When combined with the register scoreboard, zero-cycle multithreading provides a mechanism to instantly trigger an installed but dormant thread without incurring polling or interrupt overhead. The presence state of a register becomes the activation condition. When the register is updated, its presence state becomes "full" and the previously dormant thread becomes eligible to issue instructions. The MAP architecture also supports thread priorities, which are useful for dynamic optimization of multithreaded execution during a critical code section [5] and when there is a static hierarchy of importance among the threads. In the MAP chip, the priority of each thread can be set *dynamically*, allowing detailed adjustment of its relative issue frequency.

4 Pipeline Description and Operation

Each cluster employs a basic 5-stage pipeline consisting of an Instruction Fetch (IF) stage, a Register Fetch (RF) stage, a Synchronization (SZ) stage, an Execution (EX) stage, and a Writeback (WB) stage. The operation of the cluster pipeline is similar to a standard 5-stage RISC pipeline. However, as highlighted in Figure 4, there are five key differences: the addition of the SZ stage to control instruction issue and decouple pipeline operation, the replication of pipeline and register resources to support zero-cycle multithreading, the addition of a register scoreboard, the added remote-write capability through the Cluster Switch, and the absence of a Data Fetch (DF) stage. Branches have three delay slots, however, these can be filled effectively as the execution of any instruction can be predicated on the value of one of 16 CC registers. To simplify its implementation, the cluster contains no support for either register renaming or branch prediction. The following sections examine the differentiating features of the MAP pipeline in more detail.

Pipeline Overview and Operation: The single logical cluster pipeline is implemented as a distributed set of three physically separate pipelines sharing a common IF stage. As depicted in Figure 5, the instruction issue logic (SZ_CT) controls pipeline operation with a 5-bit thread-select (TSEL) signal. The assertion of one of these bits selects the corresponding thread to advance in the pipeline for one cycle.

The operation of the MAP pipeline is highly decoupled as different portions can proceed or stall independent of other stages. During single-threaded execution, an instruction in the MAP cluster pipeline proceeds in a similar fashion to the standard RISC pipeline. However, a thread stalls if one of its instructions is in the SZ stage waiting for its operands to become present. Other threads can proceed forward and issue, execute, and write-back their results independent of the stalled thread since the pipeline registers are replicated in the IF, RF and SZ stages. The top three stages (IF, RF, and SZ) of the pipeline are stalled only if all installed threads are stalled. In this scenario, the EX and WB stages can still proceed to complete all previously issued operations. Updates and bypass-writes from the Cluster Switch port, even those targeting a stalled thread, can also still occur. Only when every execution unit in the pipeline is waiting to use a shared resource, such as the Cluster Switch or the Memory Switch, do all five stages of the pipeline stall.

Instruction Fetch Stage (IF): The IF stage of the cluster is composed of a 4KB instruction cache, an autonomous prefetch engine, a multithreaded instruction queue and a multithreaded prefetch program counter buffer. Up to eight operations from each thread can be stored in the instruction queue. The replication of both the instruction queue and the program counter buffer is the main modification to the IF stage to support zero-cycle multithreading. The transfer of instruction issue decisions from the IF stage to the SZ stage requires the generation and distribution of both the AVAIL and KILL signals. The vector of AVAIL signals *optimistically* reports the availability of operations in the instruction queue for each thread. The KILL signal is the *only* required pipeline interlock and is asserted if an instruction cache miss occurs for a selected thread that previously signaled AVAIL. Also, the IF stage contains a comparator to detect branch targets which are ERRVAL's. To support intercluster interactions, the operation encodings are extended by 3 bits to allow integer and floating-point arithmetic instructions to specify the destination cluster and the target register file (IRF/FRF). These bits are passed through the subsequent stages to the EX and WB stages with minimal hardware costs.

Register Fetch (RF): While the RF stage of the MAP performs the same function as in the basic RISC processor, it requires additional hardware features to support the three explicitly parallel mechanisms. Three modifications

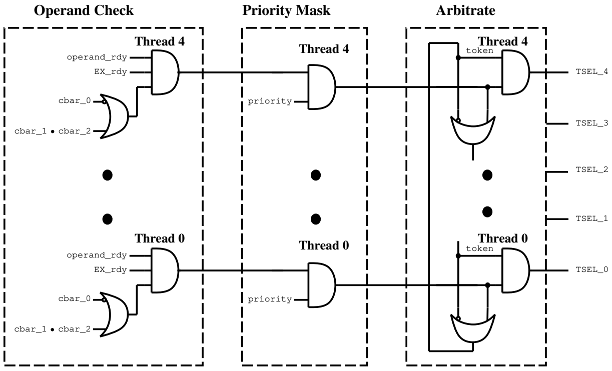

Figure 6. SZ Stage Issue Logic

are required to support intercluster interactions: the addition of a local register-scoreboard; the expansion of the normal invalidation path between the execution unit and the scoreboard, allowing the `empty` instruction to clear an arbitrary vector of register presence bits; and the addition of a Cluster Switch port to both the register files and bypass paths, enabling the identical handling of local and remote writes. Five copies of the register files and five copies of the pipeline register are required to support zero-cycle multithreading in the RF stage. Additional levels of decoding are also required in both the writeback and bypass control logic to discriminate between threads and to enable simultaneous updates by different threads to both the register files and pipeline registers. While the cluster's lock-step issue policy guarantees that local updates within a cycle are always from the same thread, remote-writes received through the Cluster Switch port can be targeted to a register in any thread slot. While no additional hardware features are needed in the RF stage to specifically support ERRVAL generation and exception deferral, all datapaths are extended by one-bit to support the guarded pointer datatype.

Synchronization Stage (SZ): The Synchronization (SZ) stage is the most significant departure from the standard RISC pipeline. Instruction issue, the zero-cycle thread-switch, and the cluster barrier are controlled by the SZ stage. The SZ stage consists of a distributed set of multithreaded reservation stations [12], one for each execution pipeline (SZ_IU, SZ_MU, SZ_FPU); a centralized pipeline controller, including the SZ_CT; and supporting bypass paths and decoding logic. The Cluster Switch port ensures that remote-writes are bypassed directly to instructions stalled in the SZ stage's reservation stations. The SZ stage also maintains a copy of the scoreboard values for all waiting instructions and reflects all modifications resulting from data updates or explicit `empty` instructions.

Each cycle, as shown in Figure 6, the SZ_CT determines which thread will issue an instruction in three steps: operand checking, priority masking, and rotating arbitration. The operand check logic determines which of the five threads are eligible to issue in the cycle. For each instruction, the SZ_CT reviews the presence of each required operand, the readiness of each requisite execution unit, the appropriate optimistic AVAIL signal, and the thread's cluster-barrier status. The SZ_CT applies the priority mask to filter the eligible threads and enforce thread priorities. The resulting eligible threads are passed to a rotating arbiter which selects the active thread for the cycle and generates an optimistic version of TSEL. If the IF stage encounters an instruction-cache miss for the selected thread, it invokes the KILL pipeline interlock, nullifying the selection and signaling the SZ stage to forego the issue slot for the cycle. An instruction is not issued if it can cause a bubble in the IF stage. Since, for each cycle, the SZ_CT logic dynamically selects from all ready threads, any ready thread may be switched into execution instantaneously, enabling zero-cycle multithreading.

The cluster-barrier logic in the SZ stage enables explicit coupling of the instruction issue decisions between clusters. The barrier is implemented by a simple finite state machine (FSM) with three states: RUN, WAIT, and SYNC. Each FSM communicates with the other two within its thread-group through a unique set of CBAR wires. Each thread slot has its own `cbar` FSM, resulting in five per cluster and a total of 15 per MAP chip.

Execution Stage (EX) and Writeback Stage (WB): The execution units in the MAP incorporate both the EX and WB pipeline stages. As all threads share the execution resources, the writeback signals between the EX and RF stage are expanded to include thread-identifiers. The execution units provide resource availability signals to the corresponding SZ stages to support issue decisions. The EX stage also trivially converts issued `cbar`'s into NOP's. As data producers need to arbitrate for the Cluster Switch one cycle in advance of data delivery, both the IU and FPU control logic are enhanced to enable arbitration requests. Also, in both the IU and FPU, the register invalidation paths to the RF stage are expanded to support explicit `empty` instructions. In addition to executing operations, the EX pipeline stage is responsible for detecting error conditions. When a deferrable error condition is detected, the EX stage automatically generates an ERRVAL to be written back in place of original operation result. The thread state is otherwise not modified. The EX stage halts the offending thread and signals an exception only when the error condition is non-deferrable. Both the decoding and multiplexing logic required to support exception detection and ERRVAL generation are also enhancements to the basic EX logic. The only required modifications to the WB stage are the added routing to the Cluster Switch for remote register writes and associated transceivers.

Module	MAP(mm^2)	Basic (mm^2)	Area
IF (w/ I$)	5.9	5.0	+ 18% (Growth)
RF	15.1	3.1	+387%
EX+WB	25.8	25.8	+ 0%
Misc	11.3	2.3	+400%
SZ	12.2	-	16% (of Cluster 0)
Cluster Switch Port	1.6	-	2%
Scoreboard/`empty`	1.4	-	2%
Guarded Pointers	3.1	-	2.7%
ERRVAL generation	0.9	-	1%
`cbar` FSM	0.04	-	0.05%
Pipelines	70.3	36.2	+ 94% (Growth)
MAP	334.0	-	-
Cluster 0	77.3	-	23% (of MAP)
Cluster 1+2	89.8	-	27%
Unified Cache	36.9	-	11%
I/O Pads	34.0	-	10%
Network + Misc	96.0	-	29%

Table 1. Logic Area Consumed

5 Evaluation of Implementation Costs

The completion of the MAP chip enables the comparison of the expected costs with the actual area and interconnect requirements. The replicated thread state required for zero-cycle multithreading was anticipated to be the most costly of the three described explicitly parallel mechanisms. While the local routing use was also expected to be noticeable, the global routing consumption was planned to be negligible. For intercluster interactions, the anticipated logic area was small and the required global routing usage was planned in detail and allocated explicitly to minimize its impact. The costs in both chip area and routing to support ERRVAL's were expected to be a minimal as well, since ERRVAL's exploit the MAP's inherent support for guarded pointers. The actual consumed area is determined by direct measurement for datapath components and by combining the synthesized netlist data with the actual random logic cell site utilization (typically $45\%-55\%$) for control logic modules. The interconnection statistics for both datapath and control modules are determined by accumulating the route lengths for all relevant nets in the wire report extracted from the place and route (P&R) chip database.

Logic Area Utilization: A summary of the measured area (square millimeters) costs is shown in the four horizontal sections of Table 1. The first section compares the area cost of zero-cycle multithreading for common pipeline stages with a hypothetical pipeline composed of single-threaded versions of the same stages. All differences are reported relative to the single-threaded stage. The second section itemizes the additional logic which, in conjunction with the previously listed components, form a complete Cluster 0. While the SZ stage requires 16% of the area, the combination of a Cluster Switch port, Scoreboard/`empty` logic and `cbar` FSM account for only 3% of the cluster. Similarly, the guarded pointer support and the added ERRVAL logic have modest requirements. All statistics are relative to Cluster 0. The third section compares the total area of the 5-stage

MAP pipeline with the hypothetical 4-stage basic pipeline and shows almost a doubling in size. However, this result overstates the penalty as the hypothetical pipeline does not include a Data Fetch stage. For reference, the fourth section provides a coarse breakdown of the contributions of each major component in the MAP. These area statistics are relative to the full chip.

Interconnect Utilization: Each of the five available routing layers in the MAP has a primary purpose. The M1 layer is used to provide routing within cells. The M5 layer distributes the pad pattern, power and clock. The M2, M3, and M4 layers are key global routing resources as they are used to interconnect cells, blocks and modules throughout the chip. The maximum potential linear routing available on the chip is 834.9Mμm and is calculated by assuming 50% coverage on each of the 5 metal layers over the full area of the chip. The analysis of the actual chip data shows that only 519.5Mμm (62%) is employed. The M2/M3/M4 usage, 252.3Mμm, represents almost 49% of the reported metalization. Automated routing for random logic interconnect and global wires predominantly employs the M2, M3, and M4 layers. It accounts for over 44% of their usage. The manual wiring for the embedded routes in full-custom macrocells, RAM arrays and also to supplement the pad pattern, clock and power distribution, accounts for the remaining 56% of consumed resources. Unfortunately, the wire report from the P&R database does not provide detailed statistics for these manual routes.

The four horizontal sections in Table 2 provides a breakdown of the interconnect usage. Each of the lengths is reported in millions of microns. The first section summarizes the contributions of manual and automated routing to both total and M2/M3/M4 interconnect usage. Only 22% of all metalization on the chip is the result of autorouting, reflecting the high degree of customization effort in the MAP's implementation. The second section highlights the resources consumed to support the explicitly parallel mechanisms. The accumulated interconnect usage is a small fraction of both the total and the M2/M3/M4 resources consumed (<5% and <8% respectively). These percentages are relative to the MAP totals. The final two sections of Table 2 enable the comparison of the local interconnect usage required within a cluster to support Cluster Switch access with the internal writeback and bypass paths. Here, the percentages are relative to the cluster total.

Analysis: Both the actual logic area and interconnect usage correlate well with the original design assumptions. Over 44% of Cluster 0 is devoted to supporting zero-cycle multithreading, confirming that the added SZ stage, routing and replicated state have a noticeable area cost. The 5-way multithreaded RF stage requires almost 400% additional area beyond the single-threaded basic version. In contrast, since

Type	M1-M5	%	M2/M3/M4	%
MAP	519.52M	100.0	252.34M	100.0
Manual Routing	407.50M	78.4	140.30M	55.6
Autorouting	112.00M	21.6	112.00M	44.4
Cluster Switch	12.02M	2.3	12.02M	4.8
zero-cycle(global)	6.50M	1.2	6.50M	2.6
ERRVAL/ptr	1.38M	0.3	1.38M	0.6
`cbar`	0.14M	–	0.14M	<0.1
Cluster 0			28.70M	100.0
IF			1.70M	5.9
RF			6.44M	22.4
SZ			5.12M	17.8
EX+WB			8.36M	29.1
Cluster Switch (local)			2.14M	7.5
Total WB/BY			1.05M	3.7
IWB			0.17M	0.6
MWB			0.11M	0.4
FWB			0.19M	0.7
IBY			0.16M	0.6
MBY			0.15M	0.5
FBY			0.27M	0.9

Table 2. Interconnect Usage(μm)

the area of the IF stage is dominated by the 4-KByte instruction cache, the effect of replication is small. Both the local usage (significant) and global routing usage (minimal) of zero-cycle multithreading also match expectations (37% and 1.2% respectively). The total added logic for fast intercluster interactions, ERRVAL's and guarded pointers is minimal, less than 8% of Cluster 0. The use of the Cluster Switch to enable intercluster interactions results in a small impact on both total routing (<3%) and M2/M3/M4 usage (<5%). The combination of guarded pointer and ERRVAL support has an insignificant effect on routing.

From the implementation, we also gain insight into organizing and correctly bypassing multiple arithmetic units. Employing a single Cluster Switch port to channel all cross-cluster operations is a significant advantage. While the local Cluster Switch consumes two times the resources used for intracluster writeback and bypassing, *all* intercluster writes from four external execution units (IU and MU from each of the other clusters) are channeled through the Cluster Switch. The use of a single Cluster Switch port provides fully general access between the independent register files and all seven arithmetic pipelines *without* the routing overhead required for the full cross-product of 49 interconnections.

6 Conclusion

The MIT MAP architecture partitions arithmetic units into clusters to localize common interactions and to allow explicit scheduling of global interactions. In existing superscalar architectures, the shared register file provides a simple unifying mechanism which implicitly performs data communication and implicitly guarantees program synchronization. With decoupled clusters and independent register files, efficient explicit mechanisms are necessary. This paper introduces three such mechanisms: *intercluster interactions*, ERRVAL based exception deferral, and *zero-cycle* multithreading. The combination of register scoreboarding, remote-writes, and the explicit `empty` instruction enable fast producer/consumer interactions. In concert, the `cbar` instruction provides a general purpose synchronization barrier amongst clusters at the cost of only one issue slot. The ERRVAL datatype encodes faulting protection and segment violations, simplifies exception deferral, and allows the elimination of unnecessary exception handling. Zero-cycle multithreading combines instantaneous thread switch, between five installed threads, with dynamic adaptation of thread selection; and thereby increases arithmetic unit utilization while preserving single-threaded performance.

There are five primary differences between the MAP cluster pipeline and the basic RISC pipeline. The novel SZ stage controls instruction issue and decouples pipeline operation. The register state is replicated in the IF, RF, and SZ stages. The local Cluster Switch port enables remote-writes and channels the delivery of remote data. Local scoreboards distribute the management of operand availability. Lastly, the Memory Switch decouples memory requests from their return values.

The analysis of the actual MAP chip design enables the assessment of the implementation impact of each of the three mechanisms. The logic required for intercluster interactions consumes only 3% of the cluster area. Intercluster interactions utilize the Cluster Switch interconnect and consume under 5% of the total M2/M3/M4 routing and under 8% of the cluster M2/M3/M4 routing. The ERRVAL features are based on the guarded pointer datatype in the MAP architecture. The combined logic area cost for both ERRVAL handling and guarded pointers is a modest 3.7% of the full cluster and has minimal routing impact. However, zero-cycle multithreading consumes 44% of the total cluster area, confirming that the replicated state is a significant cost.

The MIT Multi-ALU Processor architecture realizes effective instruction scheduling across clusters of arithmetic units by facilitating efficient intercluster interactions, enabling aggressive compiler-driven instruction scheduling and allowing effective latency tolerance. The use of a single Cluster Switch port to channel all cross-cluster operations is a significant advantage and provides fully general access while reducing interconnect and bypassing complexity. Each of the key enabling mechanisms has been successfully implemented and integrated in the cluster pipelines of the MAP chip. The resulting experience indicates that these mechanisms can also be effectively incorporated into mainstream microprocessor pipelines.

Acknowledgements

The research described in this paper was supported by the Defense Advanced Research Projects Agency under ARPA order 8272 and monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045. Thanks to the anonymous reviewers for their valuable feedback and to Sun Microsystems for their generous equipment donations. Thanks also to the Cadence Design Factory for their contributions to the physical design of the MAP chip.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, June 1990.

[2] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 319–327, Oct. 1994.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967–979, August 1988.

[4] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, MI, December 1995.

[5] S. Fiske and W. J. Dally. Thread prioritization: A thread scheduling mechanism for multiple-context parallel processors. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 210–221, Raleigh, NC, January 1995.

[6] A. Gupta and W.-D. Weber. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of 16th Annual Symposium on Computer Architecture*, pages 273–280. IEEE, May 1989.

[7] L. Gwennap. Intel, hp make epic disclosure. *Microprocessor Report*, 11(14), October 1997.

[8] S. W. Keckler and W. J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202–213, May 1992.

[9] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting fine–grain thread level parallelism on the MIT multi-ALU processor. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[10] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, August 1997.

[11] P. G. Lowney, S. G. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[12] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, January 1967.