The “MIND” Scalable PIM Architecture

Thomas Sterling and Maciej Brodowicz

Center for Advanced Computing Research, California Institute of Technology
1200 E.California Blvd., MC158-79, Pasadena, CA, USA

MIND (Memory, Intelligence, and Network Device) is an advanced parallel computer architecture for high performance computing and scalable embedded processing. It is a Processor-in-Memory (PIM) architecture integrating both DRAM bit cells and CMOS logic devices on the same silicon die. MIND is multicore with multiple memory/processor nodes on each chip and supports global shared memory across systems of MIND components. MIND is distinguished from other PIM architectures in that it incorporates mechanisms for efficient support of a global parallel execution model based on the semantics of message-driven multithreaded split-transaction processing. MIND is designed to operate either in conjunction with other conventional microprocessors or in standalone arrays of like devices. It also incorporates mechanisms for fault tolerance, real time execution, and active power management. This paper describes the major elements and operational methods of the MIND architecture.

1. INTRODUCTION

The immediate future of commercial computing is challenged by the combined trends of 1) the disparity between memory bandwidth and processor known as the “memory wall”, and 2) the need to effectively exploit multicore processor chips. Processor clock speeds continue to grow at a rate substantially greater than memory access rates, widening the gap between processor execution rates and memory delivery bandwidths. At the same time, memory chip capacity continues to track Moore’s law, increasing by about a factor of 4 every 3 years. Together, these trends are accelerating the total time measured in processor cycles required to touch every word on a memory chip. This imposes a hard barrier on the continued effective performance gain for real world applications. The migration to multicore is a response to the upper bound of effective use of increased number of transistors in single processor designs. As the number of transistors have increased in ever more complicated processor designs (e.g., Intel Itanium2) the effective number of operations per transistor has continued to decrease. At the same time, attempts to continue to increase clock rates have resulted in prohibitive power consumption while sustained performance has not improved proportionally. Multicore structures putting multiple processors on the same chip increase the number of operational ALUs without increasing the clock rate or the degree of instruction level parallelism (ILP) that the compiler needs to successfully exploit limited to single instruction stream issue. MIND is a next-generation Processor in Memory (PIM) architecture that addresses both

challenges. It exploits the very high on-chip memory bandwidth of DRAM (or SRAM) dies to attack the memory barrier while supporting a parallel model of computation through innovative mechanisms to achieve scalable computing through a potentially large array of custom multicore processors. This paper describes the MIND PIM architecture, its microarchitecture organization, its parallel instruction set and execution model, and its methods for delivering high reliability at low power.

MIND (Memory, Intelligence, and Network Device) is an advanced parallel computer architecture for high performance computing and scalable embedded processing. It is a Processor-in-Memory architecture (PIM) that exploits those semiconductor fabrication processes capable of integrating both DRAM bit cells and CMOS logic devices on the same silicon die. MIND is distinguished from other PIM architectures in that it incorporates mechanisms for efficient support of a global parallel execution model based on the semantics of message-driven multithreaded split-transaction processing. MIND is designed to operate either in conjunction with other conventional microprocessors or in standalone arrays of like devices. MIND can support conventional parallel programming practices including MPI and OpenMP, or more advanced parallel programming models being explored such as UPC and Co-Array Fortran. However, its rich support mechanisms for efficient parallel computing lends itself to new programming models that can exploit its diverse capabilities for superior efficiency and scalability. MIND reflects a global shared memory model without cache coherence. Any element of a MIND component array can directly reference any part of the system memory address space without software intervention, thus providing efficient single system image.

MIND is intended for future systems that either incorporate a very large number of components or for very long duration operation in remote regimes. To meet the reliability requirements for both extremes, MIND employs a strategy of graceful degradation for fault tolerance that allows individual elements of the MIND parallel system to fail while the rest of the system remains functional. Mechanisms for fault detection and fault isolation in the hardware are combined with runtime software for rollback, recovery, and restart of application execution. MIND is power-aware, benefiting from the low-power attributes intrinsic to the PIM while incorporating mechanisms for selectively controlling deactivation/activation of sub sections of the global parallel array to adapt power consumption to computer resource usage based on demand. Finally, limited real time response capability is provided through thread priority scheduling and guaranteed local execution time of thread operation.

The objective of this chapter is to give the reader an understanding of the MIND architecture physical organization, the basic semantic elements and governing execution model, and the principal mechanisms incorporated to support efficient parallel execution. The next section provides an overview of the MIND-based system architecture, its primary components, and their interrelationships. The most relevant examples of prior art architectures that have influenced the MIND design are discussed in Section 3. Section 4 presents a high level view of the ParalleX model of parallel execution that provides a methodology for managing computation of application concurrency on the highly replicated elements of the MIND components. Section 5 discusses the major semantic elements of the instruction set architecture. Section 6 describes the component architecture. Section 7 describes the system wide architecture of a MIND based system to define the key components and the alternative organizations that can be supported, as well as the core building block, the memory/logic “node”, which manages the execution of the application and of which many instances are

integrated on a single module (or chip). The remaining sections focus on the major individual components of the MIND node: Section 8 describes the register organization, Section 9 the wide ALU, Section 10 the thread manager, Section 11 the memory manager, and finally Section 12 the parcel handler.

2. AN OVERVIEW OF THE MIND SYSTEM ARCHITECTURE

The MIND architecture involves three levels of structure. The top level is the system, an ensemble of MIND components or "modules" and possibly other devices integrated by one or more interconnection networks. The bottom level is the "node", a bank of memory combined with the necessary logic to perform message-driven ("parcel") multithreaded execution. The intermediate level is the MIND "module" that is a collection of MIND nodes tightly connected by means of a local network and interrelated to the rest of the system by means of one or more parcel interfaces that communicate messages between modules and to other system components.

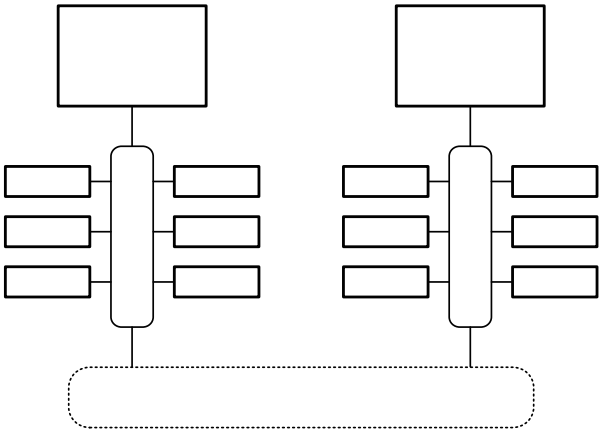

Figure 2.1. Heterogeneous system with PIMs.

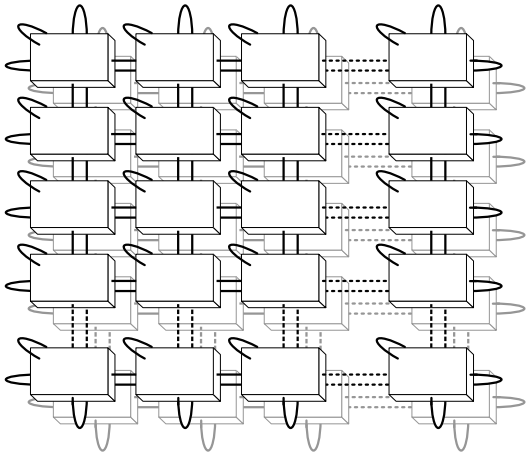

Figure 2.2. "Sea of PIMs".

MIND systems may either combine MIND modules with other classes of components (e.g., conventional microprocessors) to share the computation responsibilities or comprise an array of only MIND modules with perhaps one or more external service support processors. The former is illustrated in Figure 2.1 as a system of a cluster of microprocessors, each with a heterogeneous memory subsystem of conventional DRAM chips and MIND modules. In this class of MIND-based system, the microprocessors perform the compute intensive work and the MIND modules are allocated data-oriented work as well as some overhead tasks. The latter class of MIND-based system is sometimes referred to as a "sea of PIMs" system in which the only active component is the MIND module. This kind of system could incorporate many thousand MIND modules interconnected by a network such as a degree-6 toroidal mesh topology as shown in Figure 2.2. Here too there are conventional microprocessors in the system but only a very few and their only responsibility is to run certain parts of the operating system such as job control and the file system, providing such high level external interface services to the MIND array. In both cases, a special kind of message is employed to communicate between MIND modules. Referred to as "parcels", these messages move not only data from one module to another but also commands, dictating actions to be performed

remotely. This supports a message-driven split-transaction model of parallel computation that is intrinsically latency hiding for high efficiency.

The MIND module contains all the memory, logic, and communications interfaces required to perform coordinated parallel computation. Organized as a set of interconnected nodes, the module connects the nodes to the external parcel interfaces and to shared resources accessible to all nodes on the chip. An incident parcel will be transferred from the external parcel interface to the target node containing the destination data, invoking an action by that node related to the selected data. Similarly, the module will route a parcel instigated by any of its nodes to the appropriate parcel interface, directing it to the remote target module. The module also provides access to its shared resources by its local nodes. Different module designs can include different mixes of shared resources but may include one or more high speed pipelined floating point units, large shared instruction cache, system configuration tables, module status and control registers, and external real-time signal interface ports.

The MIND node is the core functional unit of the MIND module and system. It includes main memory, functional arithmetic units, and control logic designed to exploit the accessibility of the wide data path from the row buffer as will be shown in Figure 7.2. The MIND node is multithreaded and is message driven. Threads can be instantiated by the arrival of a parcel message from a remote MIND module. The memory employs virtual addresses locally and to all module nodes in the system. The MIND node employs a variable length binary instruction set for reduced instruction pressure. It supports multiple operations per cycle on a multi-field structure. A node can be isolated from the rest of the module in the case of failure and incorporates fault detection mechanisms in the memory, internal data paths, and some of the arithmetic function units. Highly replicated elements within the node can also be isolated from the rest of the node to permit continued operation with degraded capability even in the presence of a fault. A node clock can be reduced in rate for slower low power operation and the entire node can be powered down temporarily for active power management. The node permits limited real time execution with guaranteed response time for small local sequential tasks.

3. RELATED RESEARCH IN THE FIELD

Prior work in the disciplines of PIM and parallel computing models over the last two decades (and possibly more) has contributed greatly to the development of the MIND architecture. Here, a few highlights do inadequate justice to the rich panoply of experiences, concepts, projects, and contributors that have constituted the intellectual environment from which this work has emerged.

The idea of smart memories goes back to the days of content addressable memory and associative processors. These predated high density semiconductor integration but explored the potential performance opportunities of intimate association of logic and memory in single structures. STARAN [1] is one example of example of such architectures followed by other SIMD architectures as the Goodyear MPP [2], the MasPar MP-1 & MP-2 [3,4], and the TMC CM-2 [5].

The term PIM was coined by Ken Iobst in the late 1980s who led the IDA Terasys [6] project, another SIMD architecture with a wide row of bit level processors on the memory chip, each servicing a single column of the memory block. Peter Kogge, then at IBM, developed the Execube [7] at about the same time, which was the first MIMD PIM component, incorporating eight banks of memory, each with a dedicated processor of simple

design but independent control. Execube also had a mode in which all processors could be operated in SIMD mode from an external controller.

The IRAM [8] project at UC Berkeley led by Dave Patterson developed a PIM architecture for multimedia applications to be employed in otherwise conventional systems such as workstations and servers. The DIVA architecture [9] was developed by Draper, Hall, and others at USC ISI to provide a multicore scalable PIM architecture for a wide array of general applications including scalable embedded applications. This PIM architecture incorporated a simple mechanism for message (parcel) driven computation and supported a network that permitted the interconnection of a number of such components to work together in parallel on the same application. Two generations of the DIVA chip have been fabricated.

Message driven computation has a long history. In the late 1970s, Hewitt developed the Actor model [10], an object oriented computing model employing message-driven computation. Daly, Keckler, and Noakes at MIT developed the J-Machine [11] at MIT, a highly parallel architecture with individual processors that were message-driven. Yelick and Culler at UC Berkeley developed the active message model [12] and split-C language [13] for message driven computation for distributed memory machines through software. The DIVA architecture incorporated a variant of the parcels message driven protocol initially devised for the HTMT architecture [14]. Parcels have continued to be used as the basis for the Gilgamesh MIND architecture [15] developed by Sterling and the Cascade architecture under development by Cray Inc.

Halstead at MIT in the late 1970s developed reference trees for management of distributed virtual address spaces, possibly with copies, and later incorporated this in the early 1980s in the MultiLisp language [16] and multiprocessor implementation. Sterling has developed a variant of reference trees for address management and translation for the MIND architecture. These techniques with important advances are being employed in MIND.

The *futures* synchronization construct was also developed by Hewitt as part of the Actors model and employed very successfully by Halstead in his implementation of MultiLisp. A variant of futures was devised by Arvind in the 1980s initially at UC Irvine and then at MIT as part of the dataflow language Id Nouveau [17]. Burton Smith of Tera (now Cray) incorporated hardware mechanisms in support of futures in the MTA architecture [18] for efficient producer-consumer computation.

Multithreaded computation has a long tradition with an early implementation by Smith at Denelcor in the HEP computer [19] and then at Tera in the MTA. Gao at McGill (now University of Delaware) developed the Earth system [20] (no relation to the Japanese Earth Simulator) which was a software implementation of a multithreaded execution model. Culler at Berkeley developed the treaded abstract machine or TAM [21] for conventional multiple-processor systems.

4. PARALLEX EXECUTION MODEL

MIND departs from conventional sequential microprocessor architecture in that it is conceived from the beginning to provide for a global parallel execution model for efficient scalable computing. Clusters and MPPs use sequential processors that through software middleware support a coarse-grained distributed execution strategy of concurrent communicating sequential processes employing basic message passing that matches the distributed memory I/O-based hardware capabilities of its constituent components. In contrast, MIND based systems employ the MIND memory architecture that supports an intrinsic parallel model of computation enabling dynamic adaptive resource management,

efficient synchronization and task management, and latency hiding to effectively exploit the high degree of available memory bandwidth and logic throughput.

4.1. Shared memory

The MIND architecture supports a distributed shared memory name space. Any element of the parallel MIND system can refer to any data within the entire system directly without software intervention at the remote location of the addressed data. This is similar to the T3E and like that earlier system does not imply cache coherency. Any caching in the MIND architecture is local to the elements and memory on a given chip. The address space of the MIND architecture is virtual and is more flexible than most. Any virtually named object can be stored in any part of the system or near it depending on resource availability. A system of virtual to physical address translation is supported by the MIND architecture through a combination of hardware and software mechanisms.

The address space is partitioned into a number of distinct contexts for protection and security. Contexts are a logical resource provided by the hardware that can not be duplicated or counterfeited by software. Jobs can not touch addresses outside their own context except through an explicit protocol (beyond the scope of this paper) or between supervisor and user jobs under supervisor control. This is a limited application of capability based systems.

4.2. Continuations

The MIND parallel computing model is based on the concept of ephemeral continuations. A continuation is a set of related data that fully specifies a next computation to be performed. It must refer to some descriptive of a program and a specific entry of that descriptive that will govern the type of operation that is to be performed. It must refer to an active process (in the broadest sense) that defines the context of the computation in which the action is to be performed. Within that context, the continuation may identify argument variables upon which the action is to be performed and that may be modified as a consequence of the specified action. The continuation may also identify local or private variables accessible only to the continuation itself. A continuation is a first class object. It has a name in the global address space and it can be manipulated as such. A continuation is ephemeral. It is created at some event during the program execution and may terminate upon completion of the specific action set (not necessarily a sequence). The effect of a continuation is reflected by the change in global mutable state either directly or indirectly, including the modification to the control state of the executing program.

The MIND architecture supports three kinds of continuations that are employed for different modes of parallel flow control. The three forms of continuations are:

1. Active thread
2. Parcel
3. Lightweight control object

Although distinct in form, they represent the same basic types of information needed to govern the computation. Indeed, one form of continuation can be transformed in to one of the other forms, when circumstances warrant.

A thread is the only kind of continuation that actually causes operations to occur on a per cycle basis on MIND hardware. It is only a thread that in its own form can control MIND hardware directly for instruction issue and execute. The exception to this is that parcels can invoke atomic memory operations directly. A thread, like other continuations contains the

necessary state to govern execution of a set of actions. A thread is temporally dynamic but spatially static. Once instantiated, it exists throughout its life at a single execution site.

A parcel is the only kind of continuation that moves through a system. Once instantiated, it travels to the physical location holding the virtually named data or object that is its destination target operand. A parcel carries all of the information required to invoke a remote action. Some of those actions are primitive atomic memory operations that are performed directly on the contents of the target operand in which case the effect is direct without the need for continuation transformation. When a more complex task is to be invoked remotely, the parcel causes a thread to be instantiated at the site of the target data. The parcel designates the thread code block to be executed at the remote site and provides additional operand values that may be used for the computation. It also conveys information about the action to be performed upon its completion such as the destination variable to send a resulting value from the invoked computation. A parcel may also update the last kind of continuation, the lightweight control object.

The last form of the continuation is the lightweight control object (LCO). The LCO coordinates multiple events, conditioned on a specified criterion (or criteria) will cause an action to be performed. An LCO is not an executing entity nor does it migrate through the system. It is a smart conditional that is event driven and maintains private state between successive events, which may arrive out of order. LCOs can take on a number of forms. In fact, they can even be a snap shot of either a thread or a parcel. They serve this role when either is suspended and buffered in memory. A suspended thread is an LCO as is a suspended parcel. Although they are not performing in their normal mode, they can accept updates while suspended as LCOs and may be reactivated as a result. A couple of special LCOs are of express interest for purposes of synchronization. One is the dataflow object. This LCO, known as a *template* in the dataflow community, accepts result values from other computations and when all of the precedence constraints have been satisfied, instantiates a designated action in the form of a thread. The template LCO keeps the result value until requested, sends the result via a parcel to a specified variable, or sends it to another template LCO. Another LCO supports the *futures* construct for memory based synchronization. The futures LCO captures requests for a variable value before it has been written. When the value is finally stored, the LCO returns that value to all pending requests.

4.3. Split-transaction processing

The MIND architecture is designed to perform lightweight transaction processing. This is a dramatic departure from conventional sequential processes oriented computing. Where a conventional microprocessor-based distributed system will instantiate a single process per processor that exists for the life of the application, MIND elements process transactions in reaction to the incidence of directed requests for actions to be taken. A transaction is initiated by the arrival of such a request, conducted by the local resources on local data possibly altering the content of that data. At the termination of a transaction, one or more continuations may be generated to spawn future parallel tasks that involve, at least in part, the results of the transaction execution. In the vast majority of cases, a transaction does not make remote memory or service requests to be satisfied within its life time. While this is not true for absolutely all instances of transactions, to do so causes delays and waste of resources. Instead, such remote service requirements are satisfied by decomposing a task in to two or more tasks, one at the initiating site, and one at the remote site. This splitting of a task in two or multiple dependent components is referred to as “split transaction” processing for decoupled computation. Work is almost always performed locally and when involvement of remote data,

services, or resources is required, a new transaction is created at a remote site. When a transaction is terminated, the local hardware immediately begins to process the next pending transaction. Thus, assuming there is sufficient parallelism in the application and bandwidth within the hardware interconnection fabric, there are no delays in execution due to waiting for response from remote sites. Split transaction processing provides a powerful method of parallel system latency hiding that scales with system and application size.

Split transaction processing is enabled through the classes of continuations discussed above. An active transaction task is carried out by a thread continuation at a specific local execution site. This transaction thread was initiated by one of several events. A thread continuation can instantiate another transaction thread at the same local site. A parcel continuation can instantiate a transaction at a remote site. And, a lightweight control object continuation can cause a transaction either by instantiating a local thread or by eliciting a parcel at a remote site.

It is assumed that when a parcel is incident at a MIND execution site but there is contention for the necessary resources at that time, that some action is taken to defer the intended computation. In the case of MIND, the parcel is stored as a LCO in the local memory bank. It is converted again as a thread when resources become available. Threads also can be suspended as LCOs until resources are available to service them. In the worst case, when buffer space in memory is not available, a new parcel can be created that converts to an LCO at a remote site whose only value is that it has available space. Thus, the entire system can serve as a buffer of pending work in the form of LCOs.

5. INSTRUCTION SET ARCHITECTURE

The MIND instruction set architecture combines conventional scalar register-to-register operations employed within threads with operations for managing application and system parallelism not found in typical sequential instruction sets in support of the ParalleX model of computation. There are also a set of wide-register instructions that support multiple operations on the related fields of the wide registers to atomically manipulate structures through compound operations, called “struct processing”. In addition, auxiliary instructions provide the means for managing the system resources for fault tolerance, active power management, and real time operation. This section introduces some of the attributes and features of the MIND instruction set.

5.1. Intra-thread functional scalar instructions

A thread performs a sequence of operations. MIND is a register oriented architecture in that the arithmetic and logical functions are performed mostly on the contents of the registers. These are either register fields within the thread frame for scalar values or wide registers of other designated frames, which will be discussed in the next subsection. The thread frame fields are fixed length of 64-bits but can serve any supported data type of that size or less. These include:

- Boolean
- Logical bit vectors
- Signed and unsigned integers of 8, 16, 32, and 64 bit lengths
- IEEE 754 floating point, 32 and 64 bit format
- Address pointers

The intra-thread functional instructions are typical of the majority of RISC ISAs on these data types and include the following classes:

1. Full set of bit-wise logical operations
2. Integer add, subtract, multiply, and compare
3. Floating point add, subtract, multiply, and compare
4. Boolean operations
5. Branches and jumps

Most arithmetic operations have a test version which alters the set of condition flags. There are separate flags for positive, zero, carry, overflow, NaN, parity and additional special cases. Branches are predicated on these condition flags. A number of more common combinations of the condition codes are represented by explicit branch instructions. In addition, there is a general branch instruction with an immediate mask argument that can represent any combination of codes both in true and zero valued in Boolean sum or Boolean product relations. All instruction addresses are contiguous within a thread and checked for boundedness.

5.2. Struct processing

In addition to fields in a thread frame, the thread may reference one or more other frames within the local node. A frame may be treated as a wide register and any sub-field of that wide register may be accessed by the thread. A wide register is treated as part of the context of the thread and can hold an entire row of the memory bank. This is large enough to contain 256 bytes and hold the entire contexts of most instances of the complex data structure alternatively referred to as “records” (in Fortran) or “structs” (in C). The format of a struct is determined by user software. The wide ALU can process multiple fields simultaneously for some (but not all) operations and can perform an operation of one field of a struct conditioned on the value of another field within the same struct. This permits a number of sophisticated compound atomic actions to be performed. Examples include:

- vectors – a contiguous sequence of single typed values upon which the same operations are to be performed,
- associative searches – a set of structs, each comprising one or more data elements with a tag field, such that the value of the tag field of each struct is tested and if satisfying the criterion causes an action to be performed on one or more fields of the struct,
- in-memory synchronization – a field of one or more bits is used to manage synchronization information, either for maintaining mutual exclusion for the struct and associated data, or for general parallel flow control,
- histogramming – involving two different struct types: one a large block of equivalent structs, and the second an integer vector that holds counts of the first block for each category of a designated field,
- generic types – a data element has an associated field that specifies the type of the remainder of the struct, to determine the exact operation performed in response to a general (generic) operation. Can be extended to user defined complex data types and operation sequences,
- data driven computing – a lightweight control object that specifies an operation to be performed on arriving argument values and a destination for the result(s),
- directed graph traversal including tree-walking – with each node in the data structure represented as a struct including meta-data designating the other immediately adjacent nodes in an otherwise sparse, irregular, and possibly non-ergodic data structure,

- circular queue and stack control – control data including upper and lower bounds, head and tail offsets, empty and full condition flags (for example) of a diverse set of useful compound (but usually contiguous) data structures,
- futures synchronization – a powerful mechanism for addressing read-before-write conflicts when the consumers of a computed value do not know who the producing tasks are and vice versa.

5.3. Parallel flow control

The logical executing agent is the thread, a fixed format data structure that when allocated to one of the thread frames can cause a sequence of instruction issues by its hosting node. Instructions are provided by which a thread can be created, terminated, suspended, and synchronized. A child thread can be blocking or non-blocking but is always local to the parent thread. If a remote thread is required, i.e., a thread is to be instantiated using data at a remote site storing the target data, then instructions are used that will create an appropriate parcel. These instructions can be either implicit or explicit. An explicit parcel instruction demands the formulation of a parcel, independent of its destination and some instructions provide in-depth control of the parcel contents. Implicit instructions are generalized functional applicative commands that will create a thread if the operand is local and a parcel if the operand is remote. A set of instructions are available to test this condition to permit user application control of what to do in either case. For example, if an argument is remote, the user program (or compiler) may decide to suspend the parent thread or take other actions that could enhance overall efficiency of operation.

A set of instructions is available for the use of lightweight control objects to support this third form of continuation as described in Section 4. These include the definition of the LCO, the methods that are associated with it to control its operation. Examples, as mentioned above, include suspended threads, data flow templates, and futures. The futures is particularly important as it permits decoupled asynchronous computation with multiple producers and consumers but without prior coordination. A futures object is centered around a variable element or structure. Referencing parcels, when finding that the necessary values have been provided, treat it as a regular data element for efficient access. But when the values have yet to be committed to the location, a representation of the requesting parcel is stored locally and linked to the variable (different methods for this are used). When the requested value(s) is delivered, the pending entries in the future are reissued with the referenced value. In this way the producing and consuming tasks are synchronized without having to know about each other.

The futures construct is one example of in-memory synchronization that is used effectively by the MIND architecture and for which instructions are provided to manage the execution flow control. Most architectures use barriers sparingly and usually for coarse-grained synchronization because of the overheads involved. Some architectures have been implemented that provide hardware tag bits with supporting logic for this purpose, including older data flow architectures and the Tera MTA multithreaded architecture. MIND does not include hardware tags. But it does provide hardware and instructions for in-memory synchronization even without explicit dedicated hardware synchronization bits. Instead, these instructions operate on fields within user defined structs, providing hardware performance and low overhead with the flexibility of software defined structures. The result is a highly flexible and highly efficient near-fine grain method of parallel flow control.

6. ARCHITECTURAL DESIGN FOR POWER, RELIABILITY, AND RESPONSE

The MIND architecture incorporates additional capabilities beyond those implied by the semantics of the instruction set to provide for highly robust operation over long periods without maintenance intervention. To further the system effectiveness, the MIND architecture is also power aware to reduce average power consumption and enhance power efficiency through active power management. For certain critical code segments of embedded applications, bounded response time is essential and the MIND architecture incorporates limited real time computing on a per thread basis. These additional capabilities significantly extend the breadth of roles and dramatically reduce the risk of operation of MIND based systems.

6.1. Graceful degradation

A system characterized by single point failure modes will experience catastrophic failure (the system ceases to operate) if any of its components suffer a hard fault. Thus the mean time between failures (MTBF) of the system is a function of the mean time between failure of the components and the number of components of which it is composed. Where there are many like elements, a time versus space tradeoff can be promoted to let working subsystems fulfill the requirements of a computation even as other similar subsystems fail. The rate of computation declines as constituent elements fail, thus delivering degraded performance, but the operational lifetime of the system is substantially extended. Statistical parametric tradeoff studies have shown that MIND-like PIM organizations can achieve between three and four orders of magnitude improvement of MTBF with respect to comparable systems that exhibit single point failure modes.

In order to deliver graceful degradation of performance in the presence of faults, MIND incorporates two kinds of mechanisms. The first is isolation through reconfiguration switches. Ordinarily on, these switches can be permanently disabled through external signals to disconnect a failed memory/logic node or key duplicated elements of such nodes from the remaining system. Additional configuration state allows the control logic to operate around the missing pieces. This works well for registers, memory rows, and some redundant data paths. Arithmetic and control logic are more difficult. Because MIND memory/logic nodes incorporate wide ALUs, there are duplicate logic paths that can be exploited and time shared, assuming the permutation network is intact. This is not the case for control logic. Therefore, the fundamental Boolean formulation of the control logic is defined with the possibility of single bit or signal line errors included and the logic still finding its correct state sequence.

The second mechanism class is fault detection. Here prior art is leveraged in the typical structures of memory and data paths through error bit encoding. These are included in all hardware of the MIND components. More challenging is detecting errors in the arithmetic logic, especially transient errors. Redundant computation with scalar operations is made possible for many but not all such operations through the replicated arithmetic logic resources and additional checking logic that is included. This does require higher power and can be turned off for power conservation, a difficult tradeoff: correctness or power conservation. On a cycle available basis, background testing is performed. For the memory, this is memory scrubbing that catches bit errors early, tests the memory to determine if these are hard or soft errors and, if possible, to correct in place. Also, in background using introspective threads is a set of test suites with test vectors through the ALU and ancillary logic to check for hard faults. In spite of this aggressive mosaic of complementing mechanisms, 100% fault detection is not achieved in MIND and critical sections of the computation for which errors are unacceptable

may resort to duplicate computing on separate nodes with comparison of critical results at the end. While this is brute force, the loss of a factor of two in performance may be acceptable when orders of magnitude performance scalability is achieved.

6.2. Active power management

Power consumption is emerging as a dominant constraint on the scale and density, as well as performance and capacity, of high end computing platforms. It is additionally of considerable concern for those environments for which power is a precious resource, such as deep space missions (e.g., Mars rovers “Spirit” and “Opportunity”). For computing systems planned for the end of this decade in the low Petaflops performance regime, power budgets in excess of 10 Megawatts are anticipated, precluding their use to all but a few high-profile national laboratories.

MIND benefits from intrinsic properties of PIM resulting from several effects that combine to make the computation more power efficient. The most important factor is that operations performed in memory by local logic do not involve the external interface pins or drivers which consume much power. A second factor is that because the logic is so close to the sense amps or row buffers on the memory chip that little data movement is required reducing the on-chip data path power expended. When there is spatial locality, only one access request to a given row is required as all the data of that row can be processed without subsequent accesses to that row for the same data. Because the clock rate is approximately half that of conventional processors of the same technology generation, the energy consumed per operation is reduced as well. Generally, MIND processors are much simpler than conventional processor architectures, with approximately one tenth the numbers of gates or even less. Far fewer gates are involved in the computation thus reducing the average power consumption further. PIMs usually do not support a traditional cache layer thus eliminating that source of power demand also. Additional lesser properties of MIND also contribute to additional energy savings.

The MIND strategy for active power management employs two mechanisms of hardware control. The first provides for clock slowing. The logic of a given node has its own clocking for distribution and skew control. (Each chip has a master clock but the individual nodes even on the same semiconductor die operate asynchronously with respect to each other.) This node clock can operate at a number of speeds of factors of 2. The memory access timing control circuits can be separately adjusted as well. Only the parcel handler is maintained at full clock rate for message assimilation. Slower clocking reduces power consumption and permits low power idling when workload is low and requests are few. The second mechanism powers down MIND nodes with the exception again of the parcel handler. A node can be temporarily isolated from the rest of the MIND chip and the power cut off to stop essentially all power consumption for that node while in this state. This can also enhance long term system reliability as powered down subsystems are less likely to experience failures. While a fully shut down node will consume less power than the slowed clock, it takes much longer to restart and a local boot process must be engaged. Therefore, both mechanisms are incorporated in the MIND architecture. A third software method can temporarily discontinue certain background introspective thread processing and some redundant operations used for fault tolerance. This exposes the tradeoff between power consumption and reliability.

6.3. Real time response

For embedded computing applications responsible for sensor data assimilation and real time control of mechanical actuators, as well as some time-critical service functions in high

performance computers, bounded response time is essential. Most conventional mainstream microprocessors do not support real-time computing. MIND does, to a limited degree. Each memory/processor node can dedicate a single thread to a real time task. This thread, referred to as a *time-thread*, can be assigned to a specific I/O signal (e.g., a signal pin on a MIND chip). Except for actions triggered by catastrophic failure events, the time-thread has highest priority and guarantees action completion in bounded and predictable time. While the limitation of only a single time-thread to each node may seem over constraining, this is one of the true features of the MIND architecture. Since each MIND chip can have a substantial number of nodes and a system may comprise multiple MIND chips, each real time task can be allocated its own execution unit, ensuring that no two (or more) real-time tasks demand the same physical resources, thus avoiding any delays due to contention.

7. MIND MODULE AND MIND NODE ARCHITECTURE

In this section, we highlight the relevant features of the MIND architecture and of its constituting components, while rationalizing our design choices. We first describe the architecture of the MIND module, and then describe the architecture of the MIND nodes within a module and of the components within a MIND node.

7.1. MIND module architecture

A MIND module consists of a set of MIND nodes with accompanying interfaces and infrastructure. Such a module can be fabricated either as silicon chips or integrated further into multi-chip modules (MCMs). The number of nodes per package depends on available process technology rules and practical die sizes; the current estimates place it between 16 and 128, but we envision modules with hundreds or thousands of nodes before the middle of the next decade. The internal structure of a MIND module is depicted in Figure 7.1.

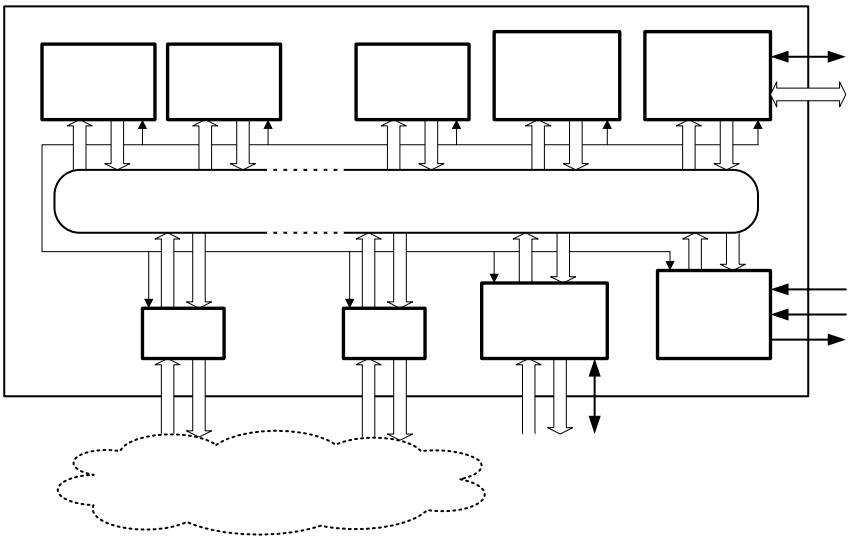

Figure 7.1. Architecture of a MIND module.

In their raw format the nodes alone cannot handle all aspects of computational tasks and communications expected. As seen in Figure 7.1, several additional subcomponents are required to provide full functionality:

- *Local Parcel Interconnect.* This is a high-bandwidth, low latency network that connects all components within a module. It is the only intra-module parcel transport medium in the module and attaches directly to parcel handlers at the nodes. This interconnect must achieve a very low latency (e.g., only a few clock cycles between issuing a request at the local node and the initialization of, for instance, the corresponding memory operation at a remote node). While low latency for accessing the functionality of a remote node is a clear requirement for nodes in the same "neighborhood" (as defined by the interconnect topology), the latency for interactions involving any two components within the module should not be much higher either. Given these requirements, although a bus-oriented topology may be sufficient for a nominal number of components, the need to alleviate contention suggests a more hierarchical organization for the local parcel interconnect.
- *Communication Ports*. These ports provide an interface between the local interconnect and the global interconnect, enabling parcel exchange between all modules. Parcels traversing the global interconnect must be "wrapped" inside packets/frames of the communication protocol proprietary to that interconnect. In contrast, the local parcel interconnect network communicates parcel content as is. The role of the ports is to facilitate parcel communication among these two networks by converting between the two parcel representations. The number of ports is typically smaller than the number of internal nodes, and is selected to satisfy the bandwidth requirements of the communication traffic incident on the module. In addition, the ports also perform buffering of messages and handle message fragmentation and reassembly in order.
- *External DRAM Interface.* This interface makes it possible to increase the available RAM capacity in the system by attaching standard "dumb" memory modules to PIM devices, thus allowing flexible platform configurations. Internally, the interface connects to the local parcel interconnect and emulates responses to remote memory access requests of a regular node. The external signaling interface conforms to industry standard protocols, such as DDR and its variants. Since the PIM nodes are capable of processing atomic memory requests locally, the interface incorporates a simplified ALU to enable this feature without undue overhead.
- *Data Streaming I/O*. This is used to communicate with external high-bandwidth streaming devices, such as mass storage (file I/O), video interfaces (cameras), or specialized processors (e.g., DSP engines). To minimize the number of dedicated external pins, most likely a form of serial, low-voltage swing differential signaling will be adopted. However, other standardized interfaces (HDMI, SATA, IEEE 1394) may be considered as well.
- *Common Functional Units.* These units complement the processing capabilities of the MIND nodes either by adding functions not directly supported by the nodes, or by implementing dedicated units to increase the performance of specific tasks. For example, if the cost of implementing a pipelined IEEE 754-compliant FPU in every node proves to be prohibitive, a number of such FPUs may be combined in a separate subcomponent, shared by all nodes.
- *Module Control Unit*. This entity monitors a number of external signal lines, processes changes in their status and distributes this information to PIM nodes and other components. The unit stores low level information describing the function and relation of the MIND module relative to the rest of the system. Besides reacting to low-level control inputs, such as global reset and interrupts, the control unit may also receive signals over a dedicated set of configurable I/O lines as well as drive them to control simple external devices (sensor arrays, mechanical actuators, etc.). Of course, different implementation

versions of MIND modules may vary the availability and the nature of configurable I/O features.

7.2. MIND node architecture

The internal structure of a MIND node embeds all the functionality necessary to provide efficient memory access, extra-node communications, and multithreaded processing. The overriding design principles aim at maintaining a high degree of autonomy of individual subcomponents as well as at maximizing local memory bandwidth, while attempting contention avoidance in component interactions. As depicted in Figure 7.2, there are five fundamental components in a MIND node:

- *Frame Cache*. This cache provides local low-latency frame storage for various key data, including thread data (active register file/stack frame), instruction stream data, auxiliary data registers (vectors and structs), runtime and system management data, and temporary data. Since the chances of access contention from various components need to be minimized, the frame cache operates with single-cycle access latency and features multiple wide data I/O buses. Additionally, the frame cache controls the allocation and deallocation of individual frames for use by other components.
- *Wide ALU*. This ALU performs permutations, arithmetic, and logical operations on data. In addition to standard processing of scalar values, the ALU can also apply SIMD-style or heterogeneous struct operations to 256-bit wide vectors of elements up to 64 bit in size each. The ALU supports both coarse-grain (element boundary) and fine-grain (bit boundary) vector element replication, permutations, and masking to take the most advantage of processing capabilities during a single pass through the ALU pipeline. To increase the effective floating-point throughput, the ALU may be augmented with a standard double-precision FPU.
- *Thread Manager*. This manager is responsible for the local execution of multithreaded code. The centerpiece of this component is a *thread scheduler* that maintains a table of active threads and selects threads for execution on a cycle-by-cycle basis, subject to resource availability, scheduling priorities, privilege level, and exception and instruction caching status. The thread manager also includes instruction fetch engines for transferring the currently executing code fragments to the frame cache, as well as execution pipelines that interface directly with the resources visible from the node and exception handler.
- *Memory Manager*. This block combines a sophisticated request handler with a fairly standard DRAM macro. Its role is fourfold: (i) handling of local memory accesses; (ii) ensuring atomicity of read-modify-write requests; (iii) internal data and metadata buffer management; (iv) application of optimization techniques, such as access combining; and (v) data replication on register boundary to comply with the intra-node bus and destination register organization.
- *Parcel Handler*. This component controls parcel traffic originating from and arriving at the node. The handler maximizes both the incoming and outgoing stream bandwidths, effectively processing a rudimentary parcel in a single cycle per stage. The receive pipeline decodes the parcel contents, extracts the data or request operands and deposits them in a pre-allocated frame registers. Conversely, the output stages can accept a proto-parcel specification residing anywhere in the frame cache or directly from the memory manager, and form and emit the outgoing parcel. Since some parcels may effect thread creation, parcel handler interfaces directly with the thread manager.

The organization of datapaths within the MIND node provides the necessary interconnect bandwidth and a high degree of independence in interfacing with internal components. At the heart of the node's floor plan resides the frame cache with multiple wide (256 bits), but relatively short unidirectional buses attached to other major components. Such an organization alleviates latencies typically associated with recharging parasitic capacitances inherently associated with long buses and eliminates the need for costly (in terms of die area and switching latency) multiplexer arrays. The access control is also vastly simplified compared to bi-directional mode. Each of the data buses can operate independently and since the requestors typically either access different frames in the cache, or the accesses to the same registers are disjoint in time, the write contentions occur with very low probability and can thus be handled by simple hardware. In this arrangement, the frame cache plays effectively the role of a high-bandwidth switch with the added benefit of single-cycle accessible storage.

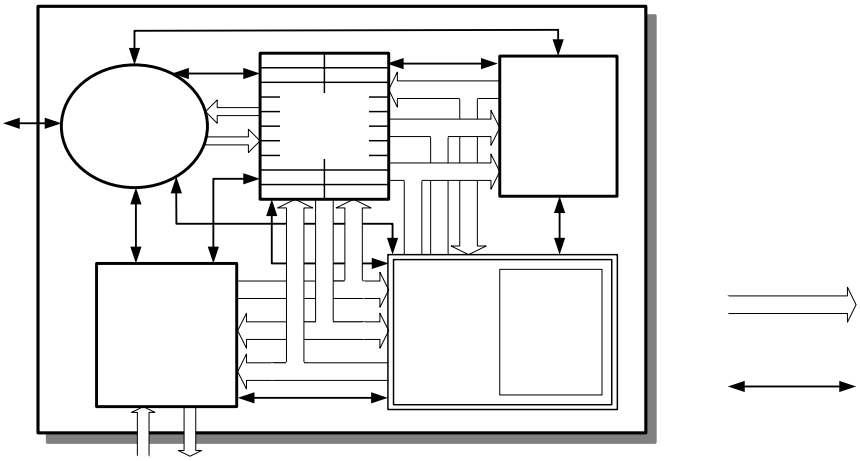

Figure 7.2. Architecture of a MIND node.

The control interfaces are routed point-to-point between the interacting entities. Any required arbitration is performed by the resource owner when multiple conflicting requests are received simultaneously. Since the amount of control information is miniscule compared to the volume of data, the use of unidirectional control buses does not pose significant problems.

In the next four sections we provide a detailed description of the Frame Cache, the Wide ALU, the Thread Manager, the Memory Manager, and the Parcel Handler.

8. FRAME CACHE

The Frame Cache is a central, register-level, instruction and data repository for the node. The storage space is partitioned into frames (2048 bit wide registers), each of which can be assigned to hold a single thread's state, cache currently accessed fragments of instruction stream, configured as temporary hardware buffer, or assigned as an auxiliary data register visible to the active threads. The frames are further subdivided into eight 256-bit wide registers, which naturally match the widths of I/O data paths and can be used to handle and transfer non-scalar data efficiently. For the purpose of standard fine-grain register access, a frame may also be viewed as a collection of 32 general purpose 64-bit registers addressable from threads. The total frame count is expected to be no lower than 64.

To minimize contention, the frame cache is multi-ported for both read and write accesses, using standard SRAM technology. The multiplexing is performed directly at bit-cell level by activating one of multiple word lines to select which of the bit lines will drive the cell's inputs for writes, or conduct the bit values stored in the cell to the sense amplifiers during reads. The only downside of this approach is the increased size of the memory cell because of additional data and control lines with associated switching transistors.

Due to the organization of wide buses, reads and writes always operate on 256-bit data chunks. However, the threads frequently require finer-grain access to registers. While the necessary alignment hardware is present in every component block connected to the frame cache outputs, the writes require only a simple replication of a scalar over 256-bit space; the target 64-bit register is selected by write control logic activating only the required subset of word lines. The frame cache also features an internal selection and replication logic attached to a dedicated pair of input and output buses, which is used to perform efficient register to register moves. This results in much improved latency of such operations compared to using the ALU and it doesn't consume any additional cycles or resources in components external to the frame cache.

There are currently two competing solutions to incorporate the instruction caching gracefully. The first assumes that each frame is equally available to be used as a data or instruction store. Hence, the OS may dynamically partition the frame cache and adjust the size of the portion allocated for instruction stream depending on the characteristics of the executing code. This approach, while flexible, potentially wastes significant die area due to multiporting. The second solution is based on the observation that since the instruction caching hardly requires multiple access buses, the optimized implementation could fit more bit cells per area unit if the code was actually stored in a dedicated, minimal I/O, structure. This has also the added benefit of removing the instruction path as another port from the data cache and offers an option of exact matching the widths of the instruction buses to the interfacing units: higher level cache on the input (capable of handling bursty traffic involving transfer of cache lines) and instruction decoder in thread manager on the output (requiring reduced width, but contiguous stream every cycle). The viability of each solution will be determined through simulation.

Besides providing physical storage, the frame cache also tracks the usage of individual frames, providing *allocate* and *deallocate* functions to the neighboring components. Since frame reservations are hardly ever performed en masse, the control automaton may be quite simple and handle such requests within a cycle. The ownership and associated responsibility to deallocate the frame when no longer in use is assigned to the original requestor, but with the possibility of OS override if problems arise.

9. WIDE ALU

Analogously to traditional processors, the MIND ALU performs all non-trivial arithmetic and logical processing on data passing through the node. Unlike many CPUs, however, it features wide operand inputs and output, extensive range of data permutations, operand masking, extended set of logical operations, unary vector, scalar-vector and vector-vector operations on many vector element sizes, and struct processing. The ALU is fully pipelined and accepts 256-bit arguments with transparent support for scalar (64-bit) operations. The vectors may be composed of elements ranging from one to eight bytes in size, packed within the 256-bit field, while scalars are right-adjusted in the rightmost 64-bit scalar field of wide operand, which complies with the data alignment applied in thread registers.

The arguments originate either from the frame cache or internal memory manager registers. The latter is necessary to implement atomic memory operations (AMOs), in which the memory has a master control over processing applied to a chunk of memory data before the result is committed back to the memory. In either case, the transfer of control is uniform and represented as a specially formed request token, naming the operation(s), argument number, types and location as well as the destination of the result. The ALU is capable of accessing the data registers of both the memory manager and the frame cache using standardized interfaces. Since control bits are decoupled from data, the tokens may be decoded before fetching the operands, which enables a convenient setup of the processing pipeline and minimizes the intermediate data buffer space. The result of processing is either a vector or a 64-bit scalar, in which case a built-in alignment network is used to adjust its location within the 256-bit output field. Besides the data outcome, the ALU generates the condition codes, which are typically stored in the thread status register by the final stages of the instruction execution pipeline, or examined directly by the requestor if the operation was triggered by an external entity, e.g., through a parcel, to determine its validity and possibly signal an exception. The condition codes are wrapped in a return token, whose additional function is to provide notification for the completion of computations. Indeed, the result write operation may be performed asynchronously without the knowledge of the requestor.

The ALU components include the coarse-grain permutation network, integer vector unit, scalar multiply-divideunit, floating point unit, and the distribution and selection network. Each of these is described in more detail below.

9.1. Coarse permutation network

The role of the coarse permutation network (CPN) includes preconditioning of the operands for the operation to be performed in the subsequent stages, rearranging the byte order in the 64-bit component subfields, and masking out the unnecessary portions of the input. The processing is performed in two largely independent pipelines, one for each of the input arguments. The operand preconditioning involves alignment of the scalar arguments, which are right-adjusted in the 256-bit field (so that the 64-bit functional units can fetch them from a predetermined subfield), and replication of scalars to form a vector of uniform elements. The latter is required for scalar-vector operations, as they are executed as vector-vector operations. The replication and alignment logic, which is organized as a set of 64-bit wide 4-way demultiplexers can also be applied to realize the coarse part of high-count bit shifts (i.e., by more than 64 positions).

The second level of permutation hardware consists of four independent modified Banyan networks, each processing a 64-bit chunk of the input vector with the 8-bit granularity in three stages. This allows an independent implementation of shifts and rotations on all four scalar fields (the final high-resolution shifting takes place in another functional block). The Banyan switch also performs arbitrary permutations and replications of vector components smaller than 8 bytes, thus reusing the same hardware structure for another task.

Finally, the output of the Banyan network is passed through the masking logic, which nullifies unwanted portions of arguments (again, on a byte boundary). Its second purpose is to provide correct sign extensions of the shifted/rotated integer vector components.

9.2. Integer vector unit

The most complex functional block of the ALU is the integer vector unit, which in turn can be subdivided into three major components: fine-grain permutation network (FPN), logical unit and vector arithmetic unit. The fine-grain network essentially helps finalize shift and

masking operations initialized in the coarse permutation unit. It consists of two stacked stages, each of which is a limited range (zero, one or two bits in either direction) shift-rotate unit combined with a masking logic. Note that superposition of FPN operations together with those of the coarse permutation network yields the full range of shift-rotate counts. The masking logic has a bit resolution and may also accept bit patterns supplied by the programmer.

The logical unit performs all typical unary (*not*) and binary bitwise (*and, or, xor, implication* with complements) operations on vectors treated as contiguous groups of bits, as well as population counts (both zeroes and ones), leading and trailing bit counts and parity in each component of the vector. This functionality is distributed across both argument's data paths, as many of these operations are mutually exclusive and require quite different processing logic. To reduce the number of logic stages, and thus the effective latency, a crossover network is used to divert operands onto secondary path when necessary.

The final processing steps in the integer vector unit are performed by a three-stage vector arithmetic unit. Besides integer adders and comparators handling argument widths of up to 64 bits, the arithmetic unit features a sophisticated reduction network, including both arithmetic and logical operations. Thanks to distribution of computing logic over both operand flows, the arithmetic unit is capable of delivering a result of bitwise logical reduction or a sum of all elements in a full vector every cycle, even if their type size is as small as byte.

9.3. Integer multiply-divide unit

The multiply-divide unit was separate from the main vector pipeline for a couple of reasons. Firstly, the latency of operations (especially division) is significantly higher than that of any elementary calculations performed in the vector pipeline. Secondly, the amount of logic implementing the desired functionality is substantial, which makes its replication to support vector operations consume rather large portion of chip die area. With the progress of process technology it is anticipated that moving at least a rudimentary multiplier to the vector unit becomes possible, while significantly lesser used functions, such as division, would be delegated to a standalone scalar unit.

The unit features two separate Wallace-tree multiply and carry-lookahead cellular array divide pipelines. Each of these operations produces 128 bits of result from the input pair of scalars, since the division yields both quotient and remainder.

9.4. Floating point unit

The FPU operates on double-precision IEEE 754 number representations. Its implementation is pipelined and supports a standard set of floating-point calculations, such as addition, subtraction, multiplication, comparison and operand conversion. More sophisticated algorithms for division and square root approximation are also planned.

9.5. Selection and distribution logic

The purpose of this final ALU stage is to identify and choose fairly the ready results from one of the parallel pipelines, and perform the data alignment before sending them to the register file. The output selection algorithm, whose scaled-down version is also used in the multiply-divide unit, provides nearly starvation-free operation with a vastly reduced level of stall back-propagation from processing pipelines to the input stages.

Each of the computing blocks described above produces results of different sizes. While the full 256-bit vectors are handled directly by the frame cache logic, scalars and 128-bit long data are replicated to be correctly written to the intended target register or register set.

10. THREAD MANAGER

The multithreaded execution model, which provides the basis for MIND programming, relies heavily on the efficient implementation and hardware support for threads. The threads are named objects, which can reside anywhere in the virtual address space. For convenience, a thread name is synonymous with the virtual address of memory holding its frame. Frames are encapsulations of the local thread state; they include contents of the register window and thread execution status with such details as current instruction pointer, condition codes, priority and privilege levels, interrupt mask, synchronization information and environment linkage. A frame occupies 2048 bits of storage (typical size of a memory row) and thus can be efficiently transferred between node's register space and memory. The frames of all threads associated with a node, executing or not, are collected in internally linked pools of memory that are pre-allocated and initialized by the operating system.

10.1. Thread management and execution

Every actively executing thread must be present in the node's frame cache and is supervised by the *thread manager*. By contrast, threads whose state has been removed from the cache, and committed to memory, are suspended. The thread manager controls all aspects of thread creation, suspension, termination, scheduling and execution, which demands a number of auxiliary tasks, such as allocation of thread entries, storage and updates of the state of active threads, instruction stream handling, monitoring resource availability, management of execution pipelines, exception processing, inter-thread synchronization, and detection and workarounds for stalls and faults. The active threads are selected for execution based on their relative priority, immediate availability of the next decoded instruction and status of the primary target resource indicated by the instruction. This eliminates priority inversion problems, in which a high-priority thread may obtain a static execution slot, but is unable to progress due to unavailability of the target resource, thus blocking an unprivileged thread. To avoid stalls inherent to a single execution pipeline dispatching requests to multiple resources with different response times, every major resource has a dedicated pipeline, which receives predecoded requests when allowed to do so by the scheduler. The optimal-FIFO-depth issue pertinent to this scenario when processing time at the resources can vary drastically is resolved by a split-phase transaction strategy. In this strategy, buffering effectively occurs directly at the resource site, or along the conduit leading to it, in a distributed fashion (e.g., in the parcel handler and interconnect buffers). Split-phase transactions also shorten the execution pipelines and their control. The pipelines dedicated to very short latency and high availability services don't need to rely on split transaction approach.

The thread manager contains a single instruction decoder for all threads; its role is to determine quickly what class of operation is to be performed and identify the target resource. Such predecoded information is stored in a relevant field of the *thread table* and retained there until the thread is scheduled for further execution. This happens when the dynamic priority value is higher than that of other active threads and the status line of the primary resource specified in the instruction signals readiness to process requests. The relevant portions of the instruction and its operand(s), including the not yet decoded fragments, are then passed to the appropriate execution pipeline for the resource. When the decoding is complete, the pipeline also generates a request token, which can be directly understood and consumed by the resource. In split-transaction pipelines, the shipping off of the token to the target execution site signifies the end of the first phase of the transaction. The second phase starts when the return token is received from the site and thus the execution pipe can learn the status of the

operation with possible exceptions incurred during the execution. At this moment, the dynamic priority of the thread is decreased (scheduling fairness policy) and the updated state information, including the new IP value and condition codes, is written to the thread's frame. Note that the instructions causing non-maskable exceptions do not perform the state write-back. Instead, their thread's entry is flagged as blocked (to remove it from the scheduler's view), relevant information is passed to an exception handler and the corresponding stage of the execution pipeline invalidated. The handler thread can analyze the information (the IP of the offending instruction can still be found in the thread's frame) and, depending on the severity of the exception, terminate the thread, suspend it, or unblock it.

Since at any time each thread has only at most one instruction being processed, the complex hazard detection and resolution circuitry known from superscalar CPUs is unnecessary. This also guarantees that instructions executed by each thread are processed in order. The threads in a group, however, may proceed at different relative speeds, affected by the response rate of resources they access and individual scheduling parameters. To allow the operating system to monitor the progress of program execution and detect potentially hazardous situations and faults, several counters capable of triggering timeout exception have been integrated with every thread's entry. Hence, if a remote processing site becomes unresponsive, this fact will eventually become known to the local runtime system. Similarly, some counters are linked to the scheduling priority computation, thus enabling reasonably efficient emulation of custom scheduling policies, or identification of cases when underprivileged threads cannot make progress.

10.2. Components of the thread manager

The functionality of the thread manager is distributed over several internal blocks:

- The *Thread Scheduler*, which maintains an internal *thread table*. The thread table contains information about active threads that is volatile and mostly invisible to the programmer. The table contains one entry per active thread, with the estimated total number of entries not exceeding 16. The thread data include, among other, the updated value of instruction pointer, indices to thread register and instruction frames, status flags (active, running, blocked, waiting for instruction, etc.), scheduling attributes (static and dynamic priorities, privilege level, timeout value, execution counter and scheduler control flags), predecoded instruction field and exception attributes. The scheduler determines which thread to run based on parallel lookup of all entries in the table. The lookup, as well as updates of the fields in thread entries take one cycle.
- The *Thread Control Unit*, which provides an external interface to the thread manager, accommodating high-level thread oriented requests such as thread creation, suspension and termination, which are produced by or relayed from other components of the MIND node. It also generates control signals to other subcomponents (particularly the thread scheduler) and coordinates them. Finally, it allocates and frees the individual frames from thread pools in memory via a dedicated free-list manager, thus mapping and unmapping thread objects in virtual namespace.
- The *Execution Engine*, which is an aggregation of all pipelines conditioning requests associated with supported resources. The engine directs output from the thread scheduler containing the next predecoded instruction to run and injects it into the relevant pipeline. The final stages of all pipelines share the bus delivering the state update data to the thread table. Currently, the supported resources include frame cache, memory manager, parcel handler, wide ALU, common functional unit, external I/O queue and external DRAM.

- The *Instruction Cache Frame Prefetch*, which initializes cache line transfers from the shared instruction repository and stores them in the frame cache. This operation is triggered as soon as the computation of the next IP during the instruction execution refers to the address outside the span of text cached in the instruction frame(s) for the thread. Since the prefetch is activated ahead of time, there exists a good chance that the new line will arrive before the next instruction is needed. Note that since cache lines and frames do not have to be of the same size, the prefetch sequence may require multiple lines to be streamed per fetch.
- The *Instruction Fetch and Predecode*, which performs two functions: it extracts individual instructions from the local instruction frame and passes them to the decoder. Compared to the frame prefetch machine, the fetch operation is much simpler (it requires one access to the frame cache, followed by an alignment step). The decoder is fairly primitive as well, since it has to determine only basic parameters of instruction execution. Both fetch automata have the authority to unblock a thread as soon as the operation completes.
- The *Exception Handler*, which has a threefold purpose: it provides an entry point for the external exceptions routed from the module control unit, it arbitrates the invocation order of the exception handlers based on predefined priorities, and it buffers parameters of simultaneously occurring exceptions. The exception handler interfaces to the final stages of the execution pipes, where the exception description returned by the executing resources may be decoded and used.
- The *Frame Cache Arbiter*, which is a minor supporting block whose function is to admit access to the frame cache to selected competing components of the thread manager. While the arbitration only minimally increases the average request turnaround time, it drastically reduces the number of supporting data buses while increasing their utilization. The arbiter caches the most recent access history internally to increase the fairness of its decisions.

11. MEMORY MANAGER

The memory manager provides the means of accessing the dynamic memory embedded in a PIM node. It services memory read and write primitives with data sizes ranging from 64-bit scalars and 256-bit vectors to 2048-bit wide memory rows/frames. To aid the PIM integration in systems employing traditional CPUs, some provisions for adjustable size cache line transfers has been made as well. Both physical and virtual addressing modes are supported. The memory manager also supervises atomic memory operations, in which a memory datum is offloaded to the wide ALU to be processed in an uninterruptible sequence. The conflicting accesses to the same memory location are guaranteed to be delayed until the result is computed and stored back. This direct support of AMOs is one of the architectural elements enabling an efficient implementation of distributed synchronization algorithms.

The design of the memory manager was driven by the need to both extract the maximum of the available memory bandwidth and provide efficient mechanisms to deal with the inherently high latency of memory accesses. The first requirement assures that the DRAM macros are utilized to their potential; the latter promotes pre-staging and early initialization of memory request processing, memory access combining (reducing the raw number of memory accesses), and efficient arbitration for multiple access channels. While the dynamic memory blocks are typically well optimized for use in standalone modules, there are possibilities of improving their efficiency in some situations based on the spatial relationship of addresses accessed in sequence. This is possible due to unhindered access to the decoder circuits in PIM. The bandwidth may also be increased by using multiple memory macros per node or

changing their internal organization; however, routing an excessive number of bit line sets and multiplexing the wide outputs of memories may prove to be too expensive in terms of space required. The variation of the last approach is to decompose a single memory block into banks that can handle the scalar data independently of each other. If scalar accesses temporarily dominate the request stream, this modification could help reclaim at least some part of the wasted bandwidth. During vector access, all address decoders and data lines remain tightly coupled.

The second set of optimizations deals with issues related to the interfacing with the requesting entities (arbitration), buffering (the internal register space has to accommodate all data supplied by the pending writes, as well as the data read from memory using dynamic allocation of buffers), request combining (where the issue is the optimal size of the working set), request processing (decoding and setup of incoming requests should overlap the memory array access as much as possible), and memory operation retirement (streaming out the results, with possible post-conditioning). While most of these are fairly straightforward, if not mundane, a clever integration of these tasks is expected to lower further the effective average memory access latency.

12. PARCEL HANDLER

The parcel handler is a communication center of the MIND node; it shapes all aspects of inbound and outbound parcel traffic. Its main functions include:

- Assimilation of parcels from the local interconnect, with the emphasis on maintaining the incoming parcel bandwidth and thus preventing the stalls of the input link. It also implies reconstruction of large parcels from elementary transfer units (flits) used directly by the communication medium.
- Parcel decoding and conversion to data aggregations understood by other node components. This involves identification of the parcel type, extraction of the local destination of the embedded request, extraction of the request itself with its arguments and repackaging of the reply address if a response to the request is expected.
- Function dispatch based on request type, which may range from a simple physical register access, through memory operations (including AMOs and page transfers) to thread instantiation. While the operations in physical space are trivial enough to be performed by the handler directly and instantly, memory and thread manager requests additionally involve register allocation and deposition of their arguments in the frame cache.
- Outgoing parcel assembly and its emission onto the interconnect. The output parcel may be generated as a result of inbound parcel processing (e.g., memory read request), or explicitly assembled by a local thread. The proto-parcel arguments supplied in each of these scenarios are different enough to require customized approaches.
- Invocation of exception handlers in case of faults or errors.
- Buffering of unprocessed parcels in the available space of the local node. While the parcel handler has only a minimal buffer space to support the request flow, it can act as a conduit and allocator to store the parcel data in the frame cache or, in the worst case, in memory. In theory, this mechanism could also be used to offload the parcel traffic to an underutilized node, should the original destination node become a communication hotspot.

The design of the parcel handler was dictated primarily by the parcel throughput requirements on both I/O links. Both input and output flows have their dedicated pipelines with a crossover bus connecting the end stage of the receive logic with the input of the

transmit pipe. The purpose of the crossover is to enable a quick route for the parcels which require minimal processing with reply, such as a physical register read, thus minimizing their turn-around time. In general, the incoming parcel traffic has a higher processing priority over the requests generated within the node. This is reasonable given that parcels are received in fragments and cannot continuously block the access to the resources from internal components. Some of the arbitration logic may therefore be simplified by not having to implement the fully qualified fairness algorithms. Analogously, the quick turn-around path is allowed to block the parcels originating from anywhere in the node when competing for the output pipeline, since otherwise the stall could back-propagate and back up the input link.

The secondary processing priority is associated with extracting the maximal memory bandwidth, and thus additional provisions have been made in the input stages to assure a quick dispatch of memory requests, such as a dedicated channel to memory manager and an auxiliary request buffer to independently retain the parcel information when arbitrating the memory access. This also alleviates the contention with other parcel-initiated actions, such as thread spawn requests, for which waiting for the preceding memory request to come through may significantly increase the latency if the memory manager is busy. Such requests rely on access to the register file only and then relinquish the control to the thread manager in a minimal number of cycles.

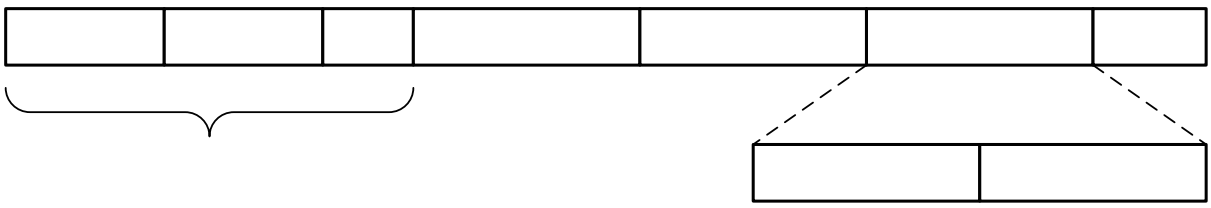

Figure 12.1. Basic parcel format.

The fundamental structure of a parcel is shown in Figure 12.1. Even parcels of this basic format can already perform a variety of actions: physical register accesses, simple thread creations, operations on scalars in memory. Frequently, all elements required to build a simple parcel can be stored within a single 256-bit datum, taking advantage of very fast transfers from the frame cache. More complex and larger parcels are formed by reusing the basic parcel's header and extending the sizes of other fields.

ACKNOWLEDGMENTS

The authors would like to express their deep gratitude to William D. Whittaker of NASA/JPL for lending his unparalleled expertise in the field of logic and VLSI design, and countless hours spent in discussions leading to the refinement of the architectural components of the MIND system. Our thanks also extend to Susan Powell, who assisted and significantly helped to shape this document into its final form. We also thank Prof. Henri Cassanova of UCSD for his substantial contributions in editing this document.

REFERENCES

1. K. Batcher, STARAN Parallel Processor System Hardware. *Proc. AFIPS Conference 43 (1974) 405-410.*

2. K. Batcher, Design of a Massively Parallel Processor. *IEEE Trans. on Computers 29:9 (1980) 836-840.*
3. T. Blank, The MasPar MP-1 Architecture. *IEEE Compcon (1990) 20-24.*
4. MasPar Corporation, Sunnyvale, California, MasPar System Overview. *Doc. 9300-0100, Rev. A3, March 1991.*
5. W. Hillis, The Connection Machine. *MIT Press, Cambridge, Mass., 1985.*
6. M. Gokhale, B. Holmes and K. Iobst, Processing In Memory: the Terasys Massively Parallel PIM Array. *IEEE Computer (1995) 23-31.*
7. P. Kogge, The EXECUBE Approach to Massively Parallel Processing. *Proc. Int. Conference on Parallel Processing 1 (1994) 77-84.*
8. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick, A Case for Intelligent RAM: IRAM. *IEEE Micro (1997) 34-44.*
9. J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. Woo Kang, I. Kim and G. Daglikoca, The Architecture of the DIVA Processing-In-Memory Chip. *Proc. ICS '02 (2002).*
10. C. Hewitt and H.G. Baker, Actors and Continuous Functionals. *Proc. IFIP Working Conference on Formal Description of Programming Concepts (1977) 367-390.*
11. M. Noakes, D. Wallach and W. Dally, The J-Machine Multicomputer: An Architectural Evaluation. *Proc. 20th Int. Symp. on Computer Architecture, 1993.*
12. T. von Eicken, D. Culler, S. Goldstein and K. Schauser, Active Messages: A Mechanism for Integrated Communication and Computation. *Proc. 19th Int. Symp. on Computer Architecture* (*1992) 256-266.*
13. D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick, Parallel Programming in Split-C. *Proc. SC99 (1999).*
14. G. Gao, K. Likharev, P. Messina and T. Sterling, Hybrid Technology Multithreaded Architecture. *Proc. 6th Symp. on the Frontiers of Massively Parallel Computation (1996) 98-105.*
15. T. Sterling and H. Zima, Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing. *Proc. SC02 (2002).*
16. R. Halstead, Jr., Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Programming Languages and Systems 7:4 (1985) 501-538.*
17. R. Nikhil, S. Pingali and Arvind, Id Nouveau. *Tech. Rep. Memo 265, Computational Structures Group, Laboratory for Computer Science, MIT, July 1986.*
18. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith, The Tera Computer System. *Proc. ICS '90 (1990) 1-6.*
19. B. Smith, Architecture and Applications of the HEP Multiprocessor Computer System. *Proc. SPIE - Real Time Signal Processing IV 298 (1981) 241-248.*
20. H. Hum, O. Maquelin, K. Theobald, X. Tian, G. Gao and L. Hendren, A Study of the EARTH-MANNA Multithreaded System. *J. Int. Parallel Programming 24 (1996) 319-347.*
21. D. Culler, S. Goldstein, K. Schauser and T. von Eicken, TAM - A Compiler Controlled Threaded Abstract Machine. *J. Parallel and Distributed Computing 18:3 (1993) 347-370.*
22. T. Sterling, J. Brockman and E. Upchurch, Analysis and Modeling of Advanced PIM Architecture Design Tradeoffs. *Proc. SC04 (2004).*