# My Memory Ain't What it Used to Be:

# Directions in Memory Interfaces

Steven Swanson

Generals Report

## 1 Introduction

> We must never allow the future to be weighed down by memory.
>
> – Milan Kundera, *The Book of Laughter and Forgetting*

The increasing disparity between memory access time and computation speed, the so-called *memory wall*, is a topic of intense research, and there is no sign that a solution is at hand.

Overcoming the memory wall is difficult, in part, because the conventional processor-memory interface just allows reads and writes and provides guarantees on the order in which they occur. There is no way to describe parallelism or potential parallelism, express temporal and spatial locality information, construct complex inter-thread data dependences, or manipulate the behavior of the memory hierarchy to match program requirements. Processors ferret out this information by observing running programs and/or speculating on their behavior, resulting in complex machines whose performance is difficult understand and reason about [1].

As the memory wall gets "taller," the importance of the memory interface only increases. A recent model [2] suggests that soon memory latency will almost completely dominate a processor's bottom-line performance. Therefore, it makes little sense to build more sophisticated processors that exploit instruction level parallelism. Researchers should instead focus on building memory-oriented processors primarily concerned with expressing, finding, and exploiting memory access locality and parallelism.

This paper investigates the state-of-the-art in memory interfaces from two perspectives, processor architectures

and programming languages. Since our goal is to devise new solutions to the memory wall, the reason for studying processors is obvious. Languages provide a rich vocabulary for describing how a program uses particular variables and interacts with memory in general. Memory-oriented designs could exploit this information to improve performance. Architectures could also borrow ideas from the interfaces languages provide to make hardware interfaces more expressive, useful, or flexible.

This work is explicitly concerned with the *interface* that a programming language or architecture defines, not the underlying implementation. For instance, although there are many cache coherence protocols [3], they all present essentially the same interface and accomplish the same task, so we discuss them only briefly. Likewise, compilers can often infer a great deal of information about memory aliasing, but the discussion here only addresses ways to express aliasing information in the language.

For space reasons, this paper ignores many interesting areas including (but not limited to) the interface to "local" storage such as registers and local variables, address protection and translation mechanisms (page tables, etc.), general hardware support for complex objects, hardware support for the stack, garbage collected languages, memory allocation, typed assembly languages, and the instruction store [1].

The next four sections describe areas where processor architectures and languages already work closely together. Section 2 describes the basic uni- and multi-processor interfaces and recent work on consistency models. Section 3 discusses transactional memories. Section 4 describes support for mutable state in functional languages, and Section 5 investigates support for manipulating streams and vectors. The next three sections describe language support for expressing memory aliasing information, spatial architectures, and processor-in-memory designs, respectively. Finally, Section 9 concludes.

## 2 The Processor-Memory Interface

Modern processor architectures must define the behavior of both a processor running in isolation and of multiple processors sharing a single memory. A processor's *consistency model* formally defines how this sharing occurs. We describe the uniprocessor interface and then discuss multiprocessor consistency models.

[1] A survey of these areas is left as a six week exercise for the reader.

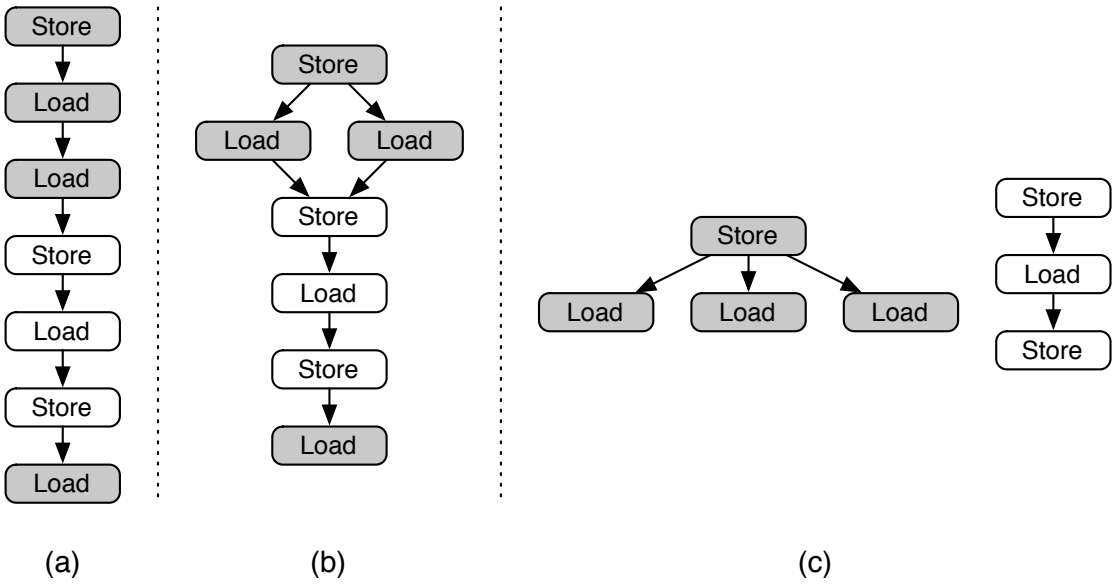



Figure 1: The total order on memory operations that program order defines (a) is unnecessarily restrictive. Allowing loads to execute in parallel (b) reveal some parallelism, but allowing memory operations to different addresses (shown in different colors) to execute in parallel can reveal even more (c).

## 2.1 The Uniprocessor Interface

We begin by describing, at a high level, the interface to memory that a single processor running in isolation provides to programs. Unless otherwise stated, the description below is of a modern, RISC-style, von Neumann microprocessor.

Most modern microprocessors provide very similar interfaces to memory. They can LOAD data from memory and STORE values into memory. LOADs typically take an address argument and produce the value read from memory. STOREs take two inputs, an address and the value to be stored, and produce no results. Different "flavors" of memory operations access different sized data items and different data types (e.g., floating point versus integer values).

A key aspect of the memory interface is the order in which memory accesses occur. Processors guarantee that memory operations appear to complete in *program order*. The program counter, guided by control instructions, defines program order by generating a sequence of instructions that the processor executes.

Although program order totally orders all memory operations, two observations reveal that memory parallelism is still possible. First, consecutive loads can execute in any order because they do not modify memory. Second, for a single processor, if two memory operations access different addresses, they can be re-ordered arbitrarily. However, since the address a LOAD or STORE accesses is only known at run-time, the processor must dynamically detect this

parallelism to exploit it. Figure 1 shows the effect these two observations have on available parallelism. The next section discusses the more complicated issue of memory ordering in multiprocessor systems.

## 2.2 Consistency

While the uniprocessor interface to memory is relatively simple, providing predictable behavior and good performance for multiple processors is much more complicated. The multiprocessor's consistency model defines a contract for how each processor observes accesses to shared memory. Most consistency models subsume the ordering constraints for a uniprocessor system (i.e., accesses by one processor to a particular address must occur in program order). They vary in the guarantees they provide for memory accesses to different addresses.

The simplest consistency model, sequential consistency, provides two guarantees. First, memory requests from a particular processor are applied to memory in the program order defined by that processor (total ordering). Second, all processors will observe memory operations to occur in the same order (sequentialization).

In practice, processors use cache coherence protocols help enforce these constraints. Coherence protocols are necessary because most systems allow each processor to have cached copies of a data value and the copies must stay synchronized. The coherence system plays two roles in enforcing a consistency model. First, it ensures that writes to a single address are seen in the same order by all processors. Second, it provides the processor with the information necessary to enforce the consistency model. For instance, in a sequentially consistent system, if a processor STOREs a value and then attempts to LOAD the value back, the LOAD cannot complete until the STORE is seen by all the other processors. This means the coherence system must invalidate (or update) any copies in other caches and inform the processor when all the invalidations are complete, so it can release the LOAD value. A rich literature on cache coherence protocols exists, but primarily addresses protocol performance, not the interface it provides.

Consistency models that make weaker guarantees than sequential consistency are called *relaxed* models. The total-ordering and sequentialization constraints are orthogonal and can be relaxed independently. Relaxed models allow more flexibility in implementations and provide higher performance, but they make it difficult to perform and reason about communication between processors.

Figure 2 demonstrates the difficulty. The two code fragments are running on separate processors and share the variables $x$ and $flag$. $B$ is waiting for $A$ to write a value to $x$. When $A$ has STORED the value in $x$ it signals $B$ by

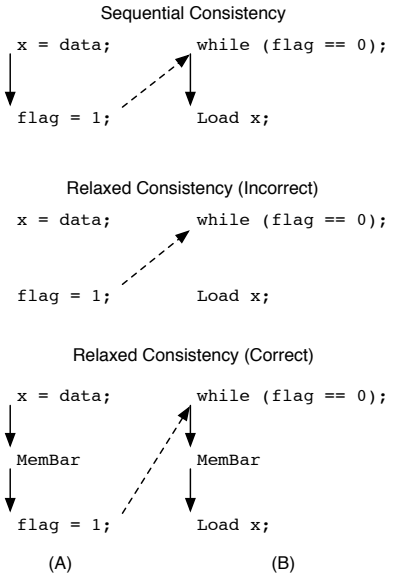



Figure 2: Two communicating threads ($A$ and $B$). Top: Correct behavior under sequential consistency. Middle: Non-deterministic behavior under a relaxed model. Bottom: Correct behavior under a relaxed model.

STOREing 1 into $flag$. In the top pane, the black arrows show the ordering constraints under sequential consistency. Both processors must observe the write to $x$ and $flag$ in the same order. The dashed line denotes that the `while` loop will not terminate until $A$'s STORE to $flag$ is visible to $B$. This data dependence, combined with sequential consistency, ensures that $A$ and $B$ will communicate successfully.

The middle scenario is the same code executing under a model that does not guarantee that memory operations from a single processor will be seen in same order by all processors. As a result, $A$'s STORE of $x$ might not be visible to $B$ until after $B$ LOADs $x$. The bottom frame remedies the situation by inserting *memory barrier* operations. Memory barrier instructions ensure that all memory operations before the barrier are complete and globally visible before the operations that follow. In this case, they reconstruct the ordering constraints in the top panel of the figure. Memory barriers come in many "flavors," but all serve essentially the same purpose. Instead of memory barriers, some consistency models provide special synchronizing LOAD and STORE operations that are implicit memory barriers. The model enforces ordering between these instructions and all others.

The performance of relaxed models is alluring, but programmers find the simplicity of sequential consistency very attractive. The differences between sequential and relaxed models is only important when processors access shared variables. Memory barriers allow the programmer to enforce sequential consistency in these regions to guarantee cor-

rectness. For maximum performance, programmers should apply the minimum number of memory barriers required to ensure that programs behave as they would under sequential consistency. Determining the minimum synchronization requirements is formally undecidable, but several notions of sufficient synchronization exist [4, 5]. Essentially, every access to a shared variable must be protected by synchronization operations (i.e., protected by locks).

An excellent survey [6] provides additional details on a range of well-known models, and the next section discusses several recently proposed models in more detail. Section 3 discusses an more radical consistency model based on transactions.

## 2.3 Making Coherence Explicit

Four recently proposed models – commit, reconcile, and fences (CRF) [7]; location consistency (LC) [8]; the Java memory model [9]; and DAG consistency [10] – have added the notion of a processor- or program-local memory to the consistency model, acknowledging that almost all modern processors include private L1 data caches.

The consistency models in the previous section, *implicit coherence models*, require that every LOAD and STORE implicitly invoke the cache coherence mechanism to ensure that the LOAD or STORE operates on the latest version of the variable. The models in this section, *explicit coherence models*, make invoking the coherence mechanism explicit. Since the implicit models differ only in their details, we first describe their common attributes and then the differences between them.

Implicit consistency models provide a COMMIT operation to explicitly make a value in a processor's local memory visible to other processors and a RECONCILE operation to request a new, possibly updated, copy of the value. Processors can execute COMMIT and RECONCILE operations on different addresses in any order, unless a memory barrier intervenes[2]. Memory barriers do not invoke the coherence system or restrict the ordering of LOAD and STORE instructions. Explicit coherence reduces the amount of communication required but makes the systems harder to reason about. For instance, because local data caches are finite, the models must permit the processor to perform COMMIT and RECONCILE operations at any time, without the program's knowledge.

From the programmers perspective, the key difference between implicit and explicit models is that explicit models do not automatically enforce ordering among accesses to different addresses. Figure 3 extends the example in Figure 2

[2][8] does not discuss fences or reordering constraints for LC within a single processor. We suspect they must provide a memory barrier mechanism or require in-order execution of all memory operations on a particular processor.

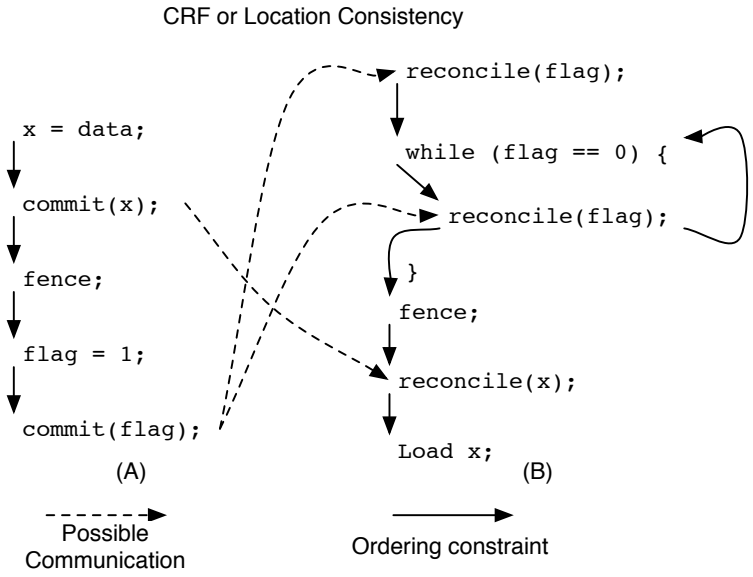



Figure 3: The communicating threads in Figure 2 implemented for an explicit coherence model.

to demonstrate. Correctness requires RECONCILE operations before (after) *every* LOAD (STORE) of a shared variable. In particular, $B$ explicitly requests the new value of $x$, instead of relying on the ordering between operations on $x$ and $flag$ that a conventional consistency model would provide.

The difference between CRF and LC is very subtle. Without COMMIT and RECONCILE instructions, there is no guarantee that writes by one processor will *ever* be seen by another. However, CRF guarantees that *if* two writes from one processor are seen by other processors, then all the processors will see them in the same order. LC allows them to be seen in different orders.

The Java memory model is similar to CRF but differs in two ways. First it provides a built-in notion of a lock. Programs can LOCK or UNLOCK a lock, and those actions are sequentially consistent. LOCK and UNLOCK operations behave as RECONCILE and COMMIT (respectively) operations for all of memory and act as memory barriers. Since LOCK and UNLOCK are the only mechanism for performing RECONCILE, COMMIT, and memory barrier operations, the Java model can only support lock-based synchronization.

One disadvantage of explicit coherence is it makes moving a thread from one processor, $P1$, to another, $P2$, difficult. To see this, consider a thread $T$ accessing a variable $V$ in the midst of a context switch. $T$, running on $P1$, STOREs to $V$. Under an implicit coherence model, the coherence system automatically acquires exclusive access to $V$

and modifies it, performing an implicit COMMIT on $V$. When the context switch occurs, a memory barrier is sufficient to ensure that the STORE to $V$ is visible to $P2$ before $T$ starts running there. Since $P1$ had exclusive access to $V$, $T$'s first access of $V$ from $P2$ will cause a cache miss and the coherence system will automatically retrieve an up-to-date version of $V$ (an implicit RECONCILE).

With explicit coherence, ensuring that $T$ is oblivious to the move from $P1$ to $P2$ change is harder. Since the COMMIT and RECONCILE operations to move $V$ from $P1$ to $P2$ are no longer automatic, the context switch might occur between a STORE to $V$ and the subsequent COMMIT. To rectify the situation, the context switch process must be conservative and COMMIT *all* values in $P1$'s local memory (i.e., write back the contents of the L1 data cache). At first glance, RECONCILing the data on $P2$ seems easier. As the thread runs, the new data will be loaded into the cache as the thread accesses it. In a correctly synchronized program, this approach will work, but if the thread that occupied $P2$ previously performed an unsynchronized write to $V$, the stale version will remain in $P2$'s cache. Ergo, $P2$'s must flush its cache before $T$ can start running.

LC and DAG consistency suffer from a second problem as well. Under these models, a RECONCILE operation must precede the first access to an address inside a critical section. This means code within a critical section, including any functions called, must be written to respect this rule. Alternatively, the processor could provide support for a "critical section mode" to automatically insert RECONCILE operations on every load, effectively changing the memory interface temporarily.

The explicit coherence models are extremely relaxed, and it is not clear that more relaxed models are possible or desirable. Several researchers [11, 12] have suggested that sequential consistency, combined with out-of-order processing and aggressive speculation is not only sufficient, but desirable since it is easy to understand. However, the increasing cost of communication means explicit coherence models may be better. The future may also lie with models that do not fit anywhere on the continuum between sequential consistency and relaxed models. The next section examines one such model.

# 3 Transactional Memory

The consistency models in Section 2 were developed to efficiently support a wide range of synchronization paradigms such as locks, semaphores, and monitors. Not coincidently, programming languages and multithreading libraries

support these mechanisms.

Recently, researchers have started to explore transactions as a general-purpose synchronization paradigm for both processors and languages. Transactions have long been used in databases to allow concurrent access to data. One likely reason for this development is the growing importance of parallelism both in workloads and in processor designs.

The key aspect of a transaction is *atomicity*: transactions (appear to) execute all at once, without interference. Transactions are attractive as a model for expressing concurrency because they have many nice properties:

- They avoid most forms of deadlock.
- They are optimistic, assuming that parallel execution is possible and rolling back as needed. This reveals more parallelism than locking, which is pessimistic.
- They are non-blocking. If one transaction stalls for some reason, others can always still make progress.
- Programmers need not choose a synchronization granularity, allowing them to use fine-grain transactions for performance critical code and coarse-grain transactions in other cases.
- They allow easy composition of thread-safe data structures.
- Research suggests that they are, by some measure, more intuitive to use than locks [13].

These benefits are not free. A transaction system must hide the effects of a transaction until it finishes and then atomically apply, or *commit*, them. Generally, this involves logging the changes the transaction makes, so they can be undone if needed. The log can potentially be of any size, so ample storage must be available for it. An implementation must also detect when two transactions conflict, so it can initiate rollback if needed. Performance is also a concern, especially in hardware-based systems, although the ease-of-use benefits transactions provide are compelling on their own.

Below, we describe the current and proposed support for transactions in processors and languages.

## 3.1 Transactions in Hardware

Hardware support for full-fledged transactional memory semantics is a relatively new idea, but architectures have provided rudimentary support in the form of COMPARE-AND-SWAP and similar instructions for some time [14].

While useful for implementing locks, semaphores, and simple lock-free data structures, COMPARE-AND-SWAP is less useful in complex scenarios (e.g., implementing complex data structures [15]).

Several researchers have proposed more extensive transactional support. Two early proposals [16, 17] provide hardware support for lock-free data structures. They provide support for transactions that read and/or modify a small number (determined by the architecture) of values, but the code for transactions must adhere to several conventions and use special memory access instructions. As a result, they are not suitable for more general applications.

Two more recent proposals allow transactions of unbounded size and allow arbitrary code. The first [18] maintains an on-chip buffer of read and/or modified data and provides hardware to allow the buffer to overflow into a hash table in main memory.

The second proposal, transactional consistency and coherence [19] (TCC), imposes a transactional model on *all* memory accesses, providing a novel consistency model and coherence protocol. Unfortunately, this approach requires a broadcast mechanism (a bus) to transmit a transaction's updates when it commits. This limits its scalability but provides a simple method for handling logs that are too long (i.e., that will not fit in the L1 cache). If a transaction's state grows too large, the processor attempts to COMMIT the portion of the transaction that has already run. If it succeeds, it "holds" the bus to prevent any other transactions from COMMITing and lets the transaction run to completion.

Finally, one group proposes transactions as a micro-architectural optimization [20]. They locate potential critical sections in hardware by looking for common locking idioms and speculatively converting the sections to transactions. To store the transactions' state they buffer cache coherence requests to allow a processor to temporarily, forcibly retain exclusive access to the affected cache lines. It is interesting to compare this approach the two above. It is substantially simpler and requires no visible changes to the architecture because it is speculative. In the worst case, it can fall back on the locks that the programmer provided. Unfortunately, it provides no help to the programmer, who still must program using error-prone locks.

## 3.2 Transaction Support in Languages

Transaction support in languages intends to make programming easier and less error prone. Performance, although obviously desirable, is a secondary concern.

Two of the systems for software transactional memory (STM) currently available are quite similar and appeared

at nearly the same time. They provide similar functionality but provide qualitatively different interfaces and make different engineering trade-offs.

The first [21] extends Java with an `atomic` keyword that designates a block of code as a transaction (similar to `synchronized` blocks). The syntax allows an optional predicate following `atomic` to specify a condition that the transaction will "block on," i.e. that must be `true` before the block will execute. To fully integrate `atomic` into Java, they discuss the semantics of `synchronized` sections and condition variables within `atomic` blocks. They also informally discuss the interaction between `atomic` sections and the Java memory model.

The second, dynamic software transactional memory (DSTM) [15], does not modify Java's syntax but provides a library for writing transactional code. In DSTM, like the previous system, execution is *obstruction-free* [22]. Obstruction freedom guarantees that a thread running in isolation will make forward progress. It forbids dead-lock but allows starvation and the possibility that two competing threads might continually interfere with one another, leading to live-lock. DSTM provides an extensible *contention manager* interface that allows programmers to easily set policy about whether and/or when one transaction can abort another.

More recent work extends on these efforts to provide a more sophisticated, *composable* transaction system (CTS) [23] in Concurrent Haskell. CTS's key contribution is a composable model for blocking transactions.

Blocking allows a thread to wait for an external event, such as a message from another thread. CTS provides two facilities to compose two expressions, $A$ and $B$, that might block. The first, sequential composition, ensures that $A$ will execute followed by $B$. The second, alternative composition, waits for exactly one of $A$ and $B$ to complete. Alternative composition requires nested transactions to isolate $A$ from $B$, and CTS is the first STM system to provide them.

Composing transactions is a powerful tool for maintaining clean abstractions. Imagine that $A$ and $B$ access abstract data types. The programmer should be able to seamlessly replace a simple, single-threaded, non-blocking implementation of $A$ with high-performance, multi-threaded, potentially blocking version. Composable transactions make this trivial. In contrast, hiding conventional locking behavior behind a clean interface is notoriously difficult.

## 3.3 Discussion

STM systems are not the first attempt to address atomicity at the language level. Researchers have used type systems to enforce both correct synchronization [24, 25, 26] and atomicity [27, 28]. Incorporating these techniques into STM systems would allow compilers to statically elide some transactions to avoid the associated overhead. The same techniques might also allow compilers to provide information to the hardware about conditions under which a transaction may be dynamically elided.

TCC, support for `atomic` blocks, and DSTM are the first steps in investigating transactional memory interfaces, and they demonstrate transactions' feasibility. CTS provides additional capabilities and constitutes a second generation STM system.

The stage is set for the second generation of transactional hardware to provide many useful features by addressing TCC's largest shortcoming: TCC is inflexible because it does not separate mechanism from policy. For instance, it does not support nested transactions, and its mechanism for blocking transactions is too primitive to efficiently support the `atomic` construct described above. Hardware support for transactional memory should be more flexible.

Decomposing transactional memory into a set of primitive capabilities that programmers could combine flexibly is one approach. Software controlled L1 caches [29, 30], general mechanisms for interposing on memory operations [31], and flexible, software-controlled address translation [32], coherence mechanisms [30] or consistency models are all mechanisms potentially useful in implementing a transactional memory system.

Finally, the TCC provides a fundamentally new model and mechanism of memory consistency. Exploring its relationship to existing models will provide insight into both transactional and conventional memory interfaces. For instance, recall that the explicit coherence consistency models in Section 2.3 specifically allow the processor to move data from processor local storage to main memory at will. This flexibility neatly avoids the problem of finite L1 data caches. To support transactions of arbitrary size, transactional memories must provide a more robust mechanism to circumvent this problem. How could a traditional consistency model exploit such a mechanism?

The purpose of transactional memories is to make changes to variables safe. The next section examines memory in languages where variables (almost) do not change at all.

# 4 Mutable State in Functional Languages

Many functional languages, such as ML and Haskell, are extremely elegant[3]. They provide a clean framework for exploring programming language concepts, while being general enough to easily express a wide range of programs.

Unlike conventional languages, such as C, functional languages traditionally disallow side-effects. In a functional language, the result of a function $f(x)$ will always have the same value for a particular value of $x$, regardless of anything else the program does. Therefore, functional languages do not allow mutable variables.

Ridding a language of mutable state has much to recommend it. Variables in purely functional languages are given a value when they are declared, and the value never changes. From the processor's perspective, only read-after-write dependencies are possible, and consistency and coherence become much simpler. Likewise, it makes reasoning about program behavior easier because values cannot change. Finally, parts of functional programs can often execute in parallel, because in the absence of an explicit data dependence, the execution of one part cannot effect the outcome of another. Despite this parallelism, functional programs remain deterministic.

Unfortunately, without side-effects difficulties arise. First, the real world is mutable. In particular, I/O operations are not side-effect-free. For instance, in the code sequence "`GetLength(f); Write(f, ''hello''); GetLength(f)`", "`GetLength(f)`" returns a different value for each call. In addition, some programming constructs, such as arrays, are difficult or inefficient to express without mutable state (see below).

In this section we examine extensions to functional languages that allow them to use mutable variables. One approach, M-structures, results in a language that is both nonfunctional and nondeterministic. The other two, I-structures and monads, preserves functionality. We discuss I-structures and M-structures first, since they are closely related.

## 4.1 I-structures and M-structures

Several functional languages, such as SISAL [33] and Id [34], exists to express parallelism and achieve high performance on scientific workloads running on dataflow processors. In brief, dataflow processors execute dataflow graphs (DFGs) directly, and no total order on instruction execution exists, only the partial order defined by the DFG. Since

[3] A positively swoony account of the perks attendant to functional programming is available at http://www.cs.washington.edu/orgs/student-affairs/cseband/studio/Enjoy%20The%20Soundness.mp3

conventional consistency models crucially depend on a total order, providing conventional memory semantics is difficult, although not impossible [35]. Dataflow languages intentionally blur the line between single- and multi-threaded programs, because they try to create highly parallel dataflow graphs. In what follows, a "thread" is a portion of a DFG that could execute in parallel with another portion.

Two difficulties arise in using functional languages for array-intensive, scientific computing. First, manipulating arrays in functional languages is inefficient because entries in the array cannot be modified after the array is created. Instead, an `update(A,i,v)` function takes array, $A$, index $i$, and value $v$ and returns a new array identical to $A$, except with $A[i] == v$. Compilers must go to great lengths to reduce the number of copies needed [36].

I-structures address this problem by providing a write-once memory structure. When an I-structure is allocated, it is *empty* and contains no value. A program is allowed to write, or fill in, an I-structure at most once. Reading from an empty I-structure blocks until the I-structure is full, while reading from a full I-structure returns the value it holds. These rules have two important implications. First, implicit synchronization makes race conditions impossible. Second, I-structures are nearly identical to normal variables in a functional language: They can be written exactly once, and their value never changes. However, their allocation and initialization can be decoupled in time.

Programs can exploit this fact to reveal parallelism. For instance, a program can allocate an array of I-structures. Then, *in parallel*, it can start filling in the values and accessing the array elements in any order. The result of the computation is identical to a program that used normal storage for the array, but the initialization and use of the array can overlap. Their designers have shown I-structures to be effective in a range of situations [37], but they do not provide general, mutable state.

M-structures [38] come closer to providing fully mutable memory, but they sacrifice the functionality. M-structures provide check-in/check-out semantics. An access to a full M-structure removes the value, and a write fills the value back in. Reads and writes are sequentialized, so only one thread can have the contents of an M-structure "checked-out" at a time.

These semantics provide an elegant solution to some problems, but they can introduce non-determinism. For instance, implementing a histogram as an array of M-structures allows parallel updates without affecting the results, since adding an element to a bucket (addition) is associative. For an non-associative operation, the results are unreliable.

I- and M-structures may have applications outside of functional languages. Both provide clean semantics for sharing data between threads, and both could be implemented efficiently as extensions to conventional memory models. After the first write to an I-structure, the consistency and coherence systems may ignore future accesses because they will be reads. A processor design could implement M-structures by allowing the CPU to hold exclusive access to a cache line.

## 4.2 Monads

*Monads* [39, 40] provide a third, more elegant solution to providing mutable state. Monads allow the programmer to specify that actions with side-effects are performed in a specific order. An example will make clear how they work. Algorithm 1 shows C-like (non-functional) code that reads two characters and prints them reverse order.

**Algorithm 1** Swapping two characters in a C-like language.

```
f = GetChar()
g = GetChar()
PutChar(g)
PutChar(f)
```

Expressing the same actions in a functional language is hard because the statements must occur in a specific order and `GetChar()` takes no arguments, so the two calls to it may execute at any time. A similar problem exists for the calls to `PutChar()`, because each depends on only one call to `GetChar()`.

**Algorithm 2** Enforcing order in a functional language.

```
// IO is the "token" for I/O activities
IO =
  let
    f = GetChar(IO)
  in
    let
      g = GetChar(f.io)
    in
      let
        tempIO = PutChar(g.io, g.c)
      in
        PutChar(tempIO, f.c)
      end let
    end let
  end let
```

Enforcing the ordering is possible but unpleasant. First, we define, by convention, a "token" called $IO$ that

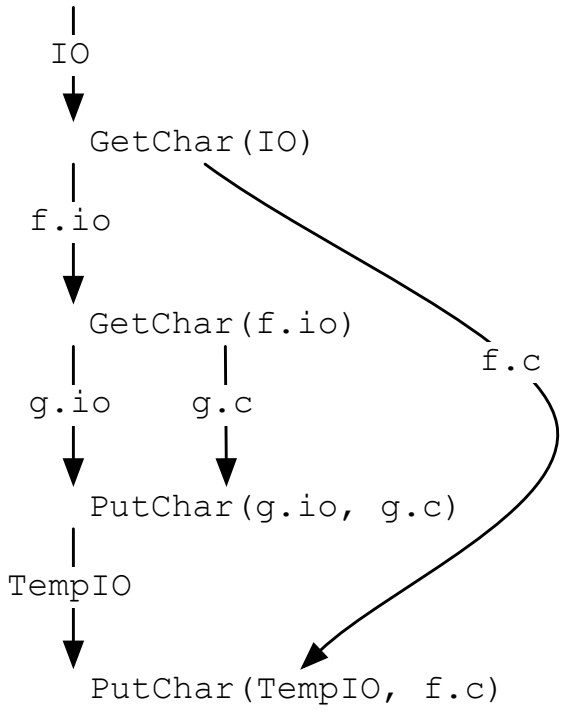



Figure 4: The data dependences in Algorithm 2. Monads allow the creation of an identical graph from Algorithm 3

represents the I/O system. Then, we modify `GetChar()` to take $IO$ as a parameter and return a data structure with two fields, $io$ and $c$, that hold the $IO$ token and the freshly read character. Finally, we modify `PutChar()` to both take the $IO$ token as an argument and return it. Algorithm 2 shows the resulting code written in a functional style. Figure 4 shows the dataflow graph for the resulting computation and reveals the sequentializing effect of the $IO$ token. Unfortunately, the code is inscrutable and writing or maintaining it is error prone, because the programmer must build ordering by hand and she has no protection from misusing the $IO$ token.

*Monads* solve both of these problems by allowing the programmer to declare that certain actions must be sequentialized. We can replace the $IO$ token in Algorithm 2 with the $IO$ monad and modify the input and return types of `GetChar()` and `PutChar()` to show that calls to them must be sequentialized by the new monad. The programmer can then write the much more comprehensible code in Algorithm 3. The `do` syntax automatically expresses the dependencies in Figure 4.

There is a rich theory behind monads, and it places some constraints on their use. In particular, a `do` block may only contain uses of a single monad. The compiler can infer which monad to use by examining the declarations of the

**Algorithm 3** Enforcing order with monads.

```
do
  f = GetChar()
  g = GetChar()
  PutChar(g)
  PutChar(f)
end do
```

functions inside it.

Monads have applications beyond I/O. They can be used to construct mutable variables, to sequentialize access to a side-effecting library, or to implement logging in a transactional memory system [23].

**Algorithm 4** Expressing parallelism with monads.

```
do
  GetInteger()
end do
+
do
  MakeRandomInteger()
end do
```

Programmers can express parallelism among sequences of monadic operations. Assume we have two monads, the $IO$ monad and a monad, $R$, that protects a random number generator. The value of a `do` block is the value of its last expression, so we can write the code in Algorithm 4 to compute the sum of the next integer on the program's input and a random number. Since each `do` block uses a different monad, the two blocks can run in parallel. If they used the same monad, they could run in either order, but their evaluation would not be interleaved [41].

Monads are very closely related to memory interfaces like CASH's [42] and WaveScalar's unordered memory interface [43] (see Section 7). Both interfaces use data dependences to enforce ordering among memory (or side-effecting) operations. Interestingly, the architectural interfaces are more general because they can easily support multiple "tokens." For instance one token could protect stack addresses while another protected access to the heap. Operations that might access either a stack or heap address would be data dependent on both tokens. Expressing these more complicated ordering constraints is beyond the power of current monad systems [41].

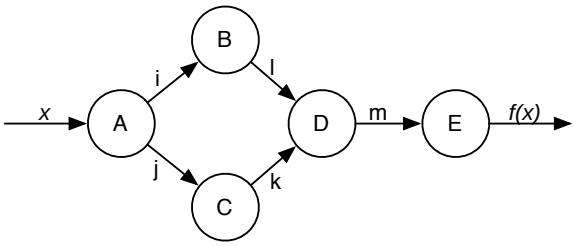



Figure 5: A simple steam program made up of five kernels ($A$-$E$) computes a stream of results, $f(x)$, from a stream of inputs, $x$. The intermediate results flow along fine streams ($i$-$m$).

# 5 Streams and Vectors

Many research projects and commercial systems have provided hardware and language support for stream and vector processing. In both cases, the high level approach is the same: expose a data structure to the programmer in the language, provide an execution model to manipulate and access it, and support the data structure and execution model in hardware.

## 5.1 Streams

*Streams* and *stream programming* have received a great deal of attention recently. Stream programming provides programming language and hardware support for a class of applications that are easily expressed as stream programs. Stream programs comprise a set of independent processes, called *kernels* or *filters*, and a set of communication channels, or *streams* of data, between them. Conceptually, a stream is an infinite sequence of data elements. The elements might be individual scalar values or more complex structures. For instance, a stream might contain pixel data from a video transmission with a red, green, and blue component or scalar samples from a digital signal.

Figure 5 shows a simple stream program that implements a function, $f$, with five kernels ($A$-$E$) connected by five streams ($i$-$m$). Inputs, $x$, enter kernel $A$ and results, $f(x)$, leave kernel $E$.

From the programmer's perspective, streams are an abstract data type, i.e., a data structure with a fixed interface. Seen this way, a stream is a FIFO queue shared by two kernels. One kernel, the *source*, may PUSH items into queue, while a second kernel, the *sink*, may POP items from the head of the queue. In addition, many algorithms are easier to express if the sink can also PEEK past the head element, but supporting PEEK complicates the interface significantly (see below). This stream interface expresses a great deal of information about how its contents may be accessed:

- Once an element is PUSHed into the stream, it can only be removed by a POPing it from head of the stream. The elements in the stream are inaccessible.
- Once an element is POPped from the stream, the stream is no longer responsible for it (i.e., the POP operation is destructive).
- Predicting which element of the stream will be accessed next is trivial: it will be the head element unless PEEK is allowed.
- The stream interface mimics a communication channel, unless PEEK is allowed.

In addition to supporting streams for communication, stream programming systems place constraints on the kernels that use them. In general, kernels should communicate with each other almost exclusively via streams, and the language may constrain the control structures the kernel can use. Below, we describe three streaming languages.

### 5.1.1 StreamC and KernelC

The StreamC and KernelC [44] languages provide the programming environment for the Imagine [45] processor and are built to be used together. KernelC is a subset of C specifically for writing kernels. It provides restricted access to global variables, only allows loop control structures, and provides some specialized support for the Imagine processor.

StreamC is based on C++ and coordinates kernel execution. It allows the creation and manipulation (copying, etc.) of streams and exposes kernels as functions that take streams as arguments.

StreamC's stream data type provide several options. Streams can be defined in terms of other streams. For instance, the program can declare a stream to contain every $n$th element of another stream. Alternatively, a stream can be declared as an *index* stream, in which the elements are addresses. In this case, PUSH operations insert an address into the stream, but POP operations remove an address, dereference it, and return the resulting value. StreamC's streams do not provide a PEEK operation.

### 5.1.2 StreamIT

The second streaming language to appear recently is StreamIt [46, 47]. StreamIt is based on Java and is tailored specifically to digital signal processing. It offers the cleanest programming model of the three languages but is more restrictive.

It constrains the topology of the connections between kernels and streams and does not allow the programmer to explicitly create streams. Programmers define kernels as Java classes and declaratively connect them. The streams are created automatically. Experience with writing signal processing programs suggests that these constraints are not onerous [47, 46].

Unlike the other languages, StreamIt supports carefully synchronized out-of-band communication between kernels. While elegant, it relies on the requirement that each StreamIt kernel always PUSHes, POPs, and PEEKs the same number of elements while processing an element. Fixing these quantities statically also allows for easier scheduling of stream traffic [46].

StreamIt stream supports the PEEK operation but the maximum number of elements a PEEK can look ahead must be bounded at compile time[4]. Constraining PEEK like this makes it easier to implement, because the kernel can implicitly buffer enough elements to satisfy the PEEK without altering the underlying stream interface.

### 5.1.3 Brook

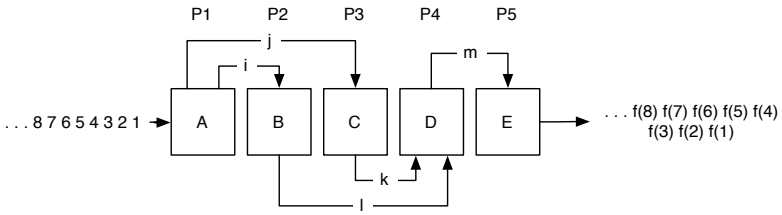
Brook [48, 49] is a streaming language targeted at scientific computing. Its kernels are similar to KernelC's, but Brook allows more general control structures and supports reduction operations (e.g., computing the sum of elements of a stream). Brook kernels are explicitly stateless.

The largest difference between Brook and other streaming languages is its definition of a stream. Brook streams can be multidimensional. For instance, a Brook stream might represent a 2x4 array of integers. Calling a kernel function on a stream implicitly applies the kernel to every element of the stream in parallel, expressing a great deal of parallelism.

This calling convention, combined with the kernels being stateless and the finite size of Brook's streams, makes Brook qualitatively different than the two streaming languages described above. Brook's streams are more like matrices or vectors than communication channels, and Brook-style kernels are basically sophisticated vector operations. Since Brook's aim is easing the translation of programs from languages like Fortran, these similarities are not surprising[5].

[4] Or so it seems. The papers do not discuss this directly, but in all the examples the argument to PEEK is either a constant or easily bounded (i.e., an index variable in a loop with constant bounds).

[5] After looking a fair amount of Brook code, it appears that in addition to its data structures, Brook has also copied a large share of Fortran's "elegance."




| | P1 | P2 | P3 | P4 | Time |
|---|---|---|---|---|---|
| Load | 1 | 2 | 3 | 4 | 1 |
| A; B; C; D; E; | | | | | 2 |
| Store | f(1) | f(2) | f(3) | f(4) | 3 |
| Load | 5 | 6 | 7 | 8 | 4 |
| A; B; C; D; E; | | | | | 5 |
| Store | f(5) | f(6) | f(7) | f(8) | 6 |

Figure 6: The program from Figure 5 occupies five processors of continuous streaming machine (top), or it can be mapped onto a chunked stream machine (bottom) with any number of processors (4 shown).

Brook supports complicated iteration patterns that many scientific applications utilize by providing *stream operators*. Stream operators let the programmer define one stream, $D$, as a structured subsets of another stream, $S$, for instance, selecting every third element, or every other column. Applying a kernel to $D$ is equivalent to writing a loop with an iteration pattern that generates $D$ from $S$ and the kernel as the body. We discuss the parallels between vector and stream computation again in Section 5.3.

## 5.2 Stream Processing Hardware

All three of the stream programming languages described in the previous section were developed with streaming processors in mind. Researchers have designed and/or built several streaming processors, Imagine [45], Merrimac [50], and others [51, 52], in recent years. Several more general-purpose architectures [53, 54, 55] provide special hardware to accelerate streaming programs. Each of these designs exploits the properties of streaming applications to reduce the amount of communication required for a computation and/or hiding communication latency. The streaming languages in the previous section are, not surprisingly, tailor-made for this purpose: the kernels only (or at least mostly) communicate via streams, and streams define a narrow interface to a communication channel.

Designing an efficient stream implementation is the key challenge in designing a streaming processor. To date,

processors use one of two solutions. Each requires the stream computation to be expressed differently to hardware. Figure 6 shows the stream program in Figure 5 mapped onto each style.

The first approach, *continuous streaming*, is the more intuitive. In continuous streaming, the kernels execute in parallel and each kernel has an execution resource dedicated to it. Stream elements flow from one kernel (processor) to another through physical communication channels (arrows in the figure). If a continuous streaming machine has $n$ processors, it can execute programs that contain up to $n$ kernels[6]. The RAW [53], SCORE [56], and many commercial streaming processors [51, 52, 56, 53] use continuous streaming.

The second approach, *chunked streaming*, divides streams into chunks that each contain $n$ stream elements, one for each processor. Each processor has a local memory with space for one or a few stream elements from each stream in the computation. Taken together, these local memories form a *streaming register file* [45] (SRF). To execute the program, the processor loads the data elements from one chunk of the input stream into the SRF (time 1 in the figure). The processing elements then execute each kernel in turn, running in SIMD fashion (time 2). As each kernel executes, it reads its inputs from the streaming register file and writes its results back. Once the kernels have completed, the machine writes the results back to main memory (time 3) and loads the next chunk of the stream (time 4). Imagine, Merrimac, and TRIPS [45, 50, 54] are chunked-streaming processors.

The two approaches represent a different set of trade-offs in streaming processor design. Chunked streaming has two key advantages.

First, chunked streaming makes moving data between kernels free (it just sits in the SRF). This enables kernels with many incoming and outgoing streams. Continuous streaming systems must physically move data from one kernel to another, and since each stream requires a communication path, high fan-in or fan-out kernels would likely lead to congestion in the processor's communication system.

Second, load balancing between kernels is unnecessary because the kernels are scheduled into a single sequence of instructions that runs in parallel on the all the processing elements. If the kernels in a continuous streaming system are not all equally complex, execution resources will sit idle. Fortunately, the problem is tractable, and the StreamIt compiler has good success load-balancing kernels for the RAW processor [46].

However, chunked streaming also suffers from several disadvantages. For instance, kernels that change the number

[6] If a program contains more than $n$ kernels, the compiler can combine multiple kernels or time-multiplex kernels across a processor.

of elements in a stream are difficult to implement, because they require writing to non-local portions of the SRF. For example, if a kernel reduces the number of elements by $1/2$, processor $n$ must write its result to the $n/2$nd slice of the SRF instead of the $n$th slice. The Imagine project describes a separate communication system to accommodate these transfers [45, 57], but they do not discuss the impact of this mechanism on complexity or cycle time. It does not appear to be implemented in their prototype [58].

Imagine leverages this communication system to allow data-dependent execution on a per-element basis. This would otherwise be difficult because the processing elements execute in SIMD fashion. Conditional streams [57] divide the stream and send elements to different kernels depending on some condition. Simple forms of conditional execution are also possible using predication.

Finally, many special-purpose streaming processors such as the MAP-CA [51] media processor provide special-purpose processing elements and stream data between them. This is not possible in a chunked streaming processor.

## 5.3 Vectors and Arrays

Vector and array processing has much longer history than stream processing, but in many ways the two are quite similar. From a language perspective vectors are also abstract data types: they are one-dimensional arrays of values. Programmers can index into vectors and read or modify their contents at will. Arrays are essentially vectors of vectors. This interface provides much less information about memory usage patterns (almost none at all) than the stream interface does.

Almost all programming languages provide support for vector and array data types, but some make them central to the language's design. These languages, such as various Fortran dialects [59, 60] and ZPL [61], are usually targeted at scientific applications where matrix arithmetic and arrays of data are commonplace. The core of these applications is typically a series of loops that operate over arrays. Often, the algorithms contain a great deal of parallelism that the programmer must reveal using the facilities that the language and the compiler provide.

High Performance Fortran (HPF) [59] provides DOALL and INDEPENDENT constructs for expressing loop-level parallelism as well as a mechanism for controlling (or at least suggesting) how data should be partitioned across processors. These features can express some parallelism, but the compiler may, with significant effort, extract even more [59].

Recently, languages like ZPL [61] have taken a different approach. ZPL is built specifically as an array manipulation language. The key abstraction in ZPL is the region. *Regions* define a set of indices or an *iteration space*. A simple region might correspond to an entire array or just its left edge. ZPL uses regions to specify the elements of an array that an operation should apply to. For instance the statement "`[R] C := A + B`" computes a pair-wise sum of the elements in `A` and `B` specified by the region `R`. Regions can be named, and region operators can define regions in terms of other regions, allowing the programmer to specify complex iteration patterns such as strided access. Regions are the *only* mechanism for accessing elements of normal arrays. Arbitrary indexing is not allowed. A special type of array allows indexing but forbids region-based access.

ZPL regions embody the same approach as Brook's derived streams. Both define multi-dimensional arrays of data or subsets thereof and make it very easy to apply operations in parallel to each element. Since both vectors and streams essentially represent sequences of independent values, this correspondence is not so surprising. From the programmer's perspective the key question is which model is the most useful. As we will see in the next section, it is easy to build a processor well suited to both styles of computation.

## 5.4 Vector Hardware

Processors with hardware support for operating on vectors have existed for many years [62] and are still being designed [63, 64]. From a vector processor's perspective, vectors are 1-dimensional arrays of scalar (integer or floating point) values. The vector processor provides instructions that operate on entire vectors at once.

Vector instruction sets allow programs to specify a long series of parallel LOAD and STORE operations with a single vector LOAD or STORE instruction. A vector processor can exploit this parallelism in several ways. Two obvious approaches correspond closely to the chunked- and continuous-streaming approaches to stream processing. The Cray-1 [62] implemented vector operations by streaming vectors through a single functional unit (continuous streaming), while the Tarantula [63] uses multiple functional units to operate on vector elements in parallel (chunked-streaming). These similarity suggest that from the hardware's perspective, the distinction between streams and vectors is largely artificial.

### 5.5 Discussion

Stream and vector processors both modify the interface to memory by allowing the programmer to specify many memory operations at once. The processor uses a state machine to issue the operations to memory and distributes the results, in effect providing a specialized *memory co-processor* (MCP[7]) for generating memory requests.

Other architectures have proposed or implemented MCPs that vary widely in interface, complexity, and aim. Decoupled access-execute architectures [65, 66] explicitly divide the instruction stream into a thread for memory access (the MCP program) and one for computation. The threads communicate via queues and run independently. Prefetching hardware mechanisms [67, 68] can be seen as simple MCPs that the running program implicitly configures. Other proposals [69, 70, 71] allow the compiler to configure spare thread contexts as a prefetching MCPs. Finally, processor-in-memory designs [72, 73, 64, 74] (see Section 8) use MCPs to improve performance by tightly integrating them with the memory itself.

These techniques share the common goal of reducing memory latency and improving performance, but other MCP applications are possible. For instance, a programmable MCP could observe an application's memory accesses to enforce application-specific protection policies for untrusted code (e.g., a web-browser plug-in), provide data-structure and access-pattern-aware prefetching for a specific application, or use aliasing information from the language (described in the next section) to optimize performance. Devising an efficient, elegant, and flexible MCP instruction set and implementation that could subsume all these specialized mechanisms is an interesting challenge.

## 6 Memory Aliasing

The problem of *memory aliasing* can be summed up as follows: it is difficult to determine at compile time if a dependence through memory could ever (or would never) exist between two memory accesses at run time. From a languages perspective, aliasing constrains the optimizations a complier can perform and makes code more difficult to reason about.

For processors, aliasing restricts flexibility in reordering memory operations. Executing two memory accesses out-of-order is only safe if they are both loads or if they access different addresses. In practice, processors circumvent

[7] Not to be confused with the Master Control Program.

this problem by using dependence predictors to guess which operations will access the same address, but speculation adds complexity, burns power, and does not scale to reordering even 10s of operations at once, since it requires checkpointing machine state.

First, we discuss currently available and proposed mechanisms for reducing pointer aliasing in languages. Then, we address the question of how to exploit aliasing information in hardware.

## 6.1 Language Support for Restricted Aliasing

Languages vary widely in the amount of aliasing they allow. At one extreme, early versions of C allowed any pointer of any type to point to any address. Languages such as Java provide much stronger protection for two reasons. First pointers are strongly typed, so it is impossible, for example, for an `int*` to point to a `float`. Second, Java constrains pointers to point to "the beginning" of objects, so if two pointers are not equal, the objects they refer do not overlap and accesses to the objects' members will not alias. This is the *no-middle-pointers* property [41].

Several proposed type systems provide even stronger assurances. For instance, linear types [75] ensure that only one pointer to a given data structure exists at one time, so aliasing is impossible. C99 [76] provides a similar, weaker facility, *restricted* pointers. Restricted pointers allow the programmer to assert that a pointer is not aliased. If this is not actually the case, behavior is undefined.

Types can provide even more sophisticated information about aliasing relationships. Ownership types [77] statically define ownership relationships between objects. Only the owner of an object may hold a pointer that object.

## 6.2 Exploiting Aliasing Information in Hardware

It is unclear how valuable aliasing information could be to conventional von Neumann microprocessors. Modern ISAs provide no mechanisms for labeling memory operations as non-aliasing. Even if they did, the benefit may be small. Memory dependence predictors are quite accurate [78], so the main savings would come from eliminating the complexity the predictor adds.

This situation may be changing. Several recent proposals [35, 79, 80, 53] allow for "instruction windows" containing 100s or 1000s of instructions. In these systems, memory dependence prediction will be insufficient to reveal the available parallelism, because the probability of correctly predicting all the dependence relationships is small.

WaveScalar [35] and RAW [53] both provide facilities for expressing aliasing information to the hardware and RAW relies upon them for good performance (see Section 7).

Hardware could also provide support the exploit the no-middle-pointers property. For instance, a set of registers would hold object addresses, and special LOAD and STORE instructions would access memory relative to one of them. Operations using different object registers could be reordered arbitrarily. The same hardware could provide parallel sequences of arbitrary memory accesses by setting the object registers to zero.

Additional approaches are certainly possible (the architectures in the next section explore several), but defining an aliasing-aware instruction set that can express the all kinds of aliasing information that languages and compilers provide is an open problem.

# 7 Spatial Architectures

Conventional superscalar designs will not continue to scale with transistor budgets. Spatial architectures are one response to this trend. Instead of building one or a few monolithic cores on a die, spatial architectures use an array of simple processing elements (PEs) and/or local memories combined with an on-chip interconnect to construct a processing substrate. The programming models and instruction sets that these architectures present are drastically different from conventional processors, and their interfaces to memory are correspondingly different. We discuss each architecture in turn and then discuss how they relate.

## 7.1 WaveScalar and ASH

WaveScalar [35] is a dataflow instruction set architecture that provides a novel interface to memory. As discussed in Section 4.1, dataflow processors do not provide a well-defined program order, making it difficult to execute conventional programs correctly. WaveScalar's *wave-ordered* interface to memory includes annotations on memory operations to encode the structure of the control flow graph within acyclic code regions called *waves*. The hardware uses this static information, combined with a dynamic notion of logical time, to reconstruct the correct memory ordering.

WaveScalar provides a second, unordered interface to memory that is unique in two ways. First, STORE instructions return a value when they have been applied to memory and are visible to all other processors. Second, the consistency model for unordered memory operations only requires that STOREs to a single address are seen in the

same order by all processors. The programmer or compiler can specifying ordering information by encoding potential memory dependences as data dependences in the program's DFG. For instance, if an unordered LOAD might access the same address as an unordered STORE, the LOAD should be made data dependent on the STORE's output. Ordered and unordered operations can freely intermingle, and a special memory barrier instruction specifies potential dependences between ordered and unordered operations.

Application Specific Hardware [42] (ASH) is a reconfigurable dataflow architecture. The hardware provides access to bare memory arrays and does not define a particular memory interface, but the current ASH tool-chain, called CASH, processes C programs and implements a memory interface similar to WaveScalar's unordered interface. The main difference is that CASH uses it for *all* memory accesses.

## 7.2 RAW

RAW [53] processors are arrays of simple scalar processors that communicate over both a statically routed, programmable network and a dynamically routed network that assures in-order delivery. The bare RAW hardware partitions memory across all the PEs and provides only a very rudimentary interface. The compiler provides the "real" interface, Maps [81], by scheduling communication on the static network and partitioning data and computation across the PEs. The Maps interface uses two types of memory instructions: static and dynamic. Static accesses are easily analyzed and use the static network.

Maps uses alias analysis to divide dynamic memory instructions into groups that *may* access the same addresses. Requests from one group of instructions flow over the dynamically routed network to a single PE that enforces the correct order and forwards them to the PEs that contains the data they need. Maps may also bypass the sequencing node and send requests directly to memory when it can determine it is safe, thereby reducing latency and increasing potential parallelism

## 7.3 Smart Memories

The goal of the Smart Memories [55] project is to provide a single device that designers can reconfigure into a special- or general-purpose processor. To that end, it provides an array of tiles that each contain a statically reconfigurable PE and set of reconfigurable memories.

The memory units can serve as caches, scratch pad memories, look-up tables, or register files. The storage arrays hold data values and user-definable control bits. The PEs access the memory cells by issuing them addresses along with a short, user-defined opcode. Specialized memory units provide content-addressable memories and multi-ported memories, and special hardware supports streaming access.

As with ASH, an unconfigured Smart Memories device is not really a processor at all and provides an extremely simple memory interface. The designer must provide all the policies and mechanisms required to execute normal programs correctly. As a measure of its success and flexibility, the Smart Memories team has demonstrated that two very different processors, Imagine [45] and Hydra [82], can be implemented on a Smart Memories device.

## 7.4 TRIPS

TRIPS [83, 54] is a dataflow-style architecture that presents a von Neumann interface. It provides an array of 64 very simple processing elements connected over an on-chip network. On one edge of the grid is a banked L1 data cache backed by a configurable L2 cache. The L2 cache is made of configurable tiles, similar to those in Smart Memories.

Depending on the configuration, the L2 tiles can serve as a conventional L2 cache, a streaming register file (SRF), or scratch pad memories. When configured as a SRF, tiles support a vector read instructions to read several consecutive values at once. To date, the Trips group has only described configurations for conventional and streaming programs, and they tailor the L2 tiles' feature set to these configurations. Additional flexibility might allow TRIPS to support more diverse programming styles.

## 7.5 Discussion

The five systems described above have several themes in common. Three of them; RAW, WaveScalar's unordered interface, and CASH; rely upon or can benefit directly from sophisticated memory aliasing information. All of the systems except WaveScalar provide either direct support streaming (RAW and TRIPS) or could be configured to support it very efficiently (ASH and Smart Memories). All of the architectures provide some degree of configurablity and attempt to exploit memory parallelism at multiple levels. These are two of the key characteristics that separate their interfaces from the others we have studied.

The third is that each architecture must provide a mechanism for enforcing order on memory requests from widely

distributed PEs. Enforcing global constraints in processors where low latency, cross-chip communication is expensive or impossible is a product of fundamental shifts in the underlying implementation technology, and solving or circumventing it is a central challenge in computer architecture.

# 8 Processor in Memory

Processor in memory (PIM) architectures tightly integrate computing resources and large amounts of memory on the same device. Researchers have proposed several PIM designs [72, 73, 64, 74], and they all attempt to achieve the same goal: dramatically increase performance by co-locating computation and the data it uses. The designs fall into two categories. IRAM proposes integrating memory onto the main processors die, perhaps obviating off-chip memory altogether. The other approaches, DIVA, ActivePages, and FlexRam, use PIM techniques to build co-processors.

IRAM [64] is a design philosophy more than a particular processor. IRAM research has focused on studying the trends in processor and memory speed, implementation technology, and power consumption to argue that integrating large amount of DRAM onto a processor die is profitable. Building an IRAM vector processor is probably worthwhile, but conventional scalar designs are unable to profitably utilize the bandwidth that large on-chip memories provide [84].

DIVA [72], FlexRAM [74], and ActivePages [73] replace conventional memory chips by presenting two interfaces. Applications that do not utilize their "intelligent" features see normal memories, while PIM-aware applications can load code onto them and trigger its execution.

The designs vary in the types computation resources they provide and the interface provided for programming them. The FlexRAM design is hierarchical and provides several processors is each memory array. DIVA provides a single scalar processor with support for vector operations. ActivePages provides, depending on the design, either reconfigurable logic [73] or a VLIW processor [85]. Notably, ActivePages provides carefully designed interfaces for both the host processor and the programmer. FlexRAM provides compiler support in the form of complicated `#pragmas`.

Interestingly, the computing resources in all the PIM designs can exploit fine-grain parallelism. A careful study of ActivePage's[85] shows that PIM processors need this ability to utilize the large bandwidth the integrated memory provides.

PIM architectures are the most general-purpose memory co-processor designs (see Section 5.5) that have been

studied, but the research to date has focused mostly on off-loading work the PIM resources. Researcher have not explored PIM's potential for providing benefits and services, such as security or user-defined memory semantics, besides improved performance.

# 9 Conclusion

The interface to memory is ripe for change for many reasons. The memory wall and the failure of conventional processor designs to continue scaling requires rethinking the processor/memory interface, and language designs must respond to the ever-increasing size and complexity of software systems. Simultaneously, both fields must grapple with novel concerns such as security, extensibility, and reliability.

Although the conventional interface that processors provide is simplistic, an enormous range of alternatives has emerged. In some cases, such as streams and vectors, researchers have already fully defined or implemented architectures that can reveal and exploit regular memory behavior. In most cases, however, the technology is immature and there are huge swathes of unexplored design space.

Architects have barely begun to examine the potential of transactional memory and spatial architectures, and work on memory-oriented processors in general is in its infancy. As researchers define and refine new interfaces to memory the largest challenge will be creating a set of primitives flexible enough to support the vast range of programming styles in use today as well as the programming paradigms of the future. Ideally, a single, comprehensive interface will emerge, providing programmers and compilers with a rich vocabulary for expressing the memory behavior of single- and multi-threaded scientific, media, integer, and commercial workloads.

To fully exploit such an interface, compilers and programming languages will need to provide powerful tools that allow programmers to succinctly express information about how programs manipulate memory. Expressive type systems are one way to provide this information, but providing language-level support for data types (such as streams or vectors) and programming paradigms is a powerful approach too. More sophisticated models for expressing and controlling fine- and coarse-grain parallelism will also be useful.

Languages and processors are "where the rubber meets the road" in the ongoing drive to increase the usefulness of computer systems, and efficient, flexible, and expressive memory interfaces are central to providing the performance and features future systems will require.

## Acknowledgments

I would like to thank Melissa Meyer, Andrew Petersen, and Andrew Schwerin for offering comments on an early draft of this paper. I would also like to thank OmniOutliner Pro, Preview, iTunes, google.com, citeseer.ist.psu.edu, and the ACM Digital library for their valiant efforts and tireless support.

## References

[1] M. Oskin, "UW CSE548 midterm," Winter 2005.

[2] S. Swanson, A. Petersen, and M. Oskin, "The death of ilp," in *ASPLOS XI Wild and Crazy Idea Session*, Oct 2004.

[3] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pp. 182–193, ACM Press, 2003.

[4] S. V. Adve and M. D. Hill, "Weak ordering—A new definition," in *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pp. 2–14, 1990.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *25 Years ISCA: Retrospectives and Reprints*, pp. 376–387, 1998.

[6] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[7] X. Shen, Arvind, and L. Rudolph, "Commit-reconcile & fences (crf): a new memory model for architects and compiler writers," in *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pp. 150–161, IEEE Computer Society, 1999.

[8] G. R. Gao and V. Sarkar, "Location consistency-a new memory model and cache consistency protocol," *IEEE Transactions on Computers*, vol. 49, pp. 798–813, August 2000.

[9] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison Wesley, 1996.

[10] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-consistent distributed shared memory," in *Proc. of the 10th Int'l Parallel Processing Symp. (IPPS'96)*, pp. 132–141, 1996.

[11] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is sc + ilp = rc?," in *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pp. 162–171, IEEE Computer Society, 1999.

[12] M. D. Hill, "Multiprocessors should support simple memory-consistency models," *IEEE Computer*, vol. 31, no. 8, pp. 28–34, 1998.

[13] C. Flanagan and S. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *In Proceedings of POPL, 2003.*, 2003.

[14] "Ibm system/370 extended architecture principles of operation." Order no. SA22-7085, available through IBM branch offices.

[15] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pp. 92–101, ACM Press, 2003.

[16] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the oklahoma update," *IEEE Parallel Distrib. Technol.*, vol. 1, no. 4, pp. 58–71, 1993.

[17] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[18] S. Lie, *Hardware Support for Unbounded Transactional Memory*. PhD thesis, Massachusetts Institute of Technology, 2004.

[19] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, p. 102, IEEE Computer Society, 2004.

[20] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 5–17, ACM Press, 2002.

[21] T. Harris and K. Fraser, "Language support for lightweight transactions," 2003.

[22] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, p. 522, IEEE Computer Society, 2003.

[23] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy, "Composable memory transactions," in *submitted to Priciples and Practice of Parallel Programming 2005*, 2004.

[24] C. Flanagan and M. Abadi, "Types for safe locking," in *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pp. 91–108, Springer-Verlag, 1999.

[25] C. Boyapati and M. Rinard, "A parameterized type system for race-free Java programs," in *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (Tampa Bay, FL), October 2001.

[26] D. Grossman, "Type-safe multithreading in cyclone," in *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pp. 13–25, ACM Press, 2003.

[27] C. Flanagan and S. Qadeer, "Types for atomicity," in *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pp. 1–12, ACM Press, 2003.

[28] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *In Proceedings of the ACM SIGPLAN 2003.*, 2003.

[29] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pp. 107–116, ACM Press, 2000.

[30] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen, "The vmp multiprocessor: initial experience, refinements, and performance evaluation," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 410–421, 1988.

[31] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, “Informing memory operations: Providing memory performance feedback in modern processors,” in *ISCA*, pp. 260–270, 1996.

[32] B. Jacob and T. Mudge, “Software-managed address translation,” in *HPCA ’97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA ’97)*, p. 156, IEEE Computer Society, 1997.

[33] D. J. Raymond, “Sisal: A safe and efficient language for numerical calculations,” *Linux J.*, vol. 2000, no. 80es, p. 14, 2000.

[34] R. Nikhil, “The parallel programming language id and its compilation for parallel machines,” in *Proceedings of the Workshop on Mazzive Paralleism: Hardware, Programming and Applications*, Acamedic Press, 1990.

[35] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, “WaveScalar,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 291, 2003.

[36] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, “A report on the sisal language project,” *J. Parallel Distrib. Comput.*, vol. 10, no. 4, pp. 349–366, 1990.

[37] Arvind, R. Nikhil, and K. K. Pingali, “I-structures: Data structures for parallel computing,” *ACM Transaction on Programming Languages and Systems*, vol. 11, no. 4, pp. 598–632, 1989.

[38] P. S. Barth, R. S. Nikhil, and A. Nikhil, “M-structures: extending a parallel, non-strict, functional language with state,” in *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pp. 538–568, Springer-Verlag New York, Inc., 1991.

[39] P. Wadler, “Monads for functional programming,” in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp. 24–52, Springer-Verlag, 1995.

[40] S. P. Jones, “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell,” in *Engineering theories of software construction*, pp. 47–96, 2001.

[41] Personal communication with Dan Grossman.

[42] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," *SIGPLAN Not.*, vol. 39, no. 11, pp. 14–26, 2004.

[43] S. Swanson, A. Schwerin, A. Petersen, M. Oskin, and S. Eggers, "Multigranular thread support in WaveScalar." Unpublished.

[44] P. Mattson, *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[45] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine stream processor," in *Proceedings 2002 IEEE International Conference on Computer Design*, pp. 282–288, Sept. 2002.

[46] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[47] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Computational Complexity*, 2002.

[48] W. J. Dally, "Stream programming languages brook tutorial," May 2002. Powerpoint.

[49] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[50] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with streams," in *SC'03*, (Phoenix, Arizona), November 2003.

[51] "Map-ca datasheet," June 2001. Equator Technologies.

[52] J. Stokes, "Introducing the ibm/sony/toshiba cell processor." http://arstechnica.com/articles/paedia/cpu/cell-1.ars.

[53] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Argarwal, "Baring it all to software: Raw machines," *IEEE Computer*, 1997.

[54] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger, "Universal mechanisms for data-parallel architectures," in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 303, IEEE Computer Society, 2003.

[55] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: a modular reconfigurable architecture," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pp. 161–171, ACM Press, 2000.

[56] E. Caspi, R. Huang, Y. Markovskiy, J. Yeh, J. Wawrzynek, , and A. DeHon, "A streaming multi-threaded model," December 2001.

[57] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 159–170, ACM Press, 2000.

[58] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the imagine stream architecture," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, p. 14, IEEE Computer Society, 2004.

[59] "High performance fortran: http://www.cs.rice.edu/ chk/hpf-tutorial.html," April 1996.

[60] C. Research, "Fortran language reference manual." http://www.cray.com/craydoc/manuals/007-3694-003/html-007-3694-003/index.html.

[61] L. Snyder, *A Programmers Guide to ZPL*. The MIT Press, 1999.

[62] R. L. Sites, "An analysis of the cray-1 computer," in *ISCA '78: Proceedings of the 5th annual symposium on Computer architecture*, pp. 101–106, ACM Press, 1978.

[63] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, "Tarantula: a vector extension to the alpha architecture," *SIGARCH Comput. Archit. News*, vol. 30, no. 2, pp. 281–292, 2002.

[64] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[65] J. E. Smith, “Decoupled access/execute computer architectures,” in *ISCA ’82: Proceedings of the 9th annual symposium on Computer Architecture*, pp. 112–119, IEEE Computer Society Press, 1982.

[66] W. A. Wulf, “The wm computer architecture,” *SIGARCH Comput. Archit. News*, vol. 16, no. 1, pp. 70–84, 1988.

[67] J.-L. Baer and T.-F. Chen, “An effective on-chip preloading scheme to reduce data access penalty,” in *Supercomputing ’91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 176–186, ACM Press, 1991.

[68] A. Roth, A. Moshovos, and G. S. Sohi, “Dependence based prefetching for linked data structures,” *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 115–126, 1998.

[69] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, “A study of slipstream processors,” in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 269–280, ACM Press, 2000.

[70] D. Kim and D. Yeung, “Design and evaluation of compiler algorithms for pre-execution,” in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 159–170, ACM Press, 2002.

[71] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, “Simultaneous subordinate microthreading (ssmt),” in *ISCA ’99: Proceedings of the 26th annual international symposium on Computer architecture*, pp. 186–195, IEEE Computer Society, 1999.

[72] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, “The architecture of the diva processing-in-memory chip,” in *ICS ’02: Proceedings of the 16th international conference on Supercomputing*, pp. 14–25, ACM Press, 2002.

[73] M. Oskin, F. T. Chong, and T. Sherwood, “Active pages: a computation model for intelligent memory,” in *ISCA ’98: Proceedings of the 25th annual international symposium on Computer architecture*, pp. 192–203, IEEE Computer Society, 1998.

[74] “Flexram: Toward an advanced intelligent memory system,” in *ICCD ’99: Proceedings of the 1999 IEEE International Conference on Computer Design*, p. 192, IEEE Computer Society, 1999.

[75] H. G. Baker, “”use-once” variables and linear objects: storage management, reflection and multi-threading,” *SIGPLAN Not.*, vol. 30, no. 1, pp. 45–52, 1995.

[76] A. S. Numerical C Extensions Group, “Restricted pointers in c,” 1994.

[77] D. G. Clarke, J. M. Potter, and J. Noble, “Ownership types for flexible alias protection,” in *OOPSLA ’98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 48–64, ACM Press, 1998.

[78] G. Z. Chrysos and J. S. Emer, “Memory dependence prediction using store sets,” in *ISCA ’98: Proceedings of the 25th annual international symposium on Computer architecture*, pp. 142–153, IEEE Computer Society, 1998.

[79] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler, “A design space evaluation of grid processor architectures,” in *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 40–51, IEEE Computer Society, 2001.

[80] S. C. Goldstein and M. Budiu, “Nanofabrics: spatial computing using molecular electronics,” in *ISCA ’01: Proceedings of the 28th annual international symposium on Computer architecture*, pp. 178–191, ACM Press, 2001.

[81] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, “Maps: a compiler-managed memory system for raw machines,” in *ISCA ’99: Proceedings of the 26th annual international symposium on Computer architecture*, pp. 4–15, IEEE Computer Society, 1999.

[82] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, “The stanford hydra cmp,” *IEEE Micro*, vol. 20, no. 2, pp. 71–84, 2000.

[83] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. H. a nd Doug Burger, S. W. Keckler, and C. R. Moore, “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture,” in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

[84] D. A. Bowman and L. F. Hodges, “Evaluation of existing architectures in iram systems,” in *Workshop on Mixing Logic and DRAM*, 1997.

[85] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. Farrens, and A. Chopra, “Exploiting ilp in page-based intelligent memory,” in *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pp. 208–218, IEEE Computer Society, 1999.