Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures

Atsuhiro Goto* Akira Matsumoto[†] Evan Tick[‡]
Institute for New Generation Computer Technology (ICOT)[§]

Abstract

This paper describes the design and performance of a tightly-coupled shared-memory coherent cache optimized for the execution of parallel logic programming architectures. The cache utilizes a copy-back write-allocation protocol having five states and a hardware lock mechanism. Optimizations for logic programming are introduced in four software-controlled memory access commands: direct-write, exclusive-read, read-purge, and read-invalidate. In this paper we describe these operations and present simulated measurements showing their performance advantage for an architecture of the committed-choice language KL1. The cache optimizations also improve the performance of non-committed-choice languages, such as OR-parallel Prolog. A version of the cache design described here is currently being implemented for ICOT's Parallel Inference Machine (PIM).

1 Introduction

Current interest in parallel logic programming stems from its declarative semantics which facilitate writing and debugging programs and remove most of the need for explicit uncovering and control of parallelism. Parallel logic programming languages, many based on Prolog, are high-level in the sense that destructive assignment is forbidden and programs often appear as many small recursive procedures instead of fewer, large iterative ones. In addition, some languages retain Prolog's nondeterminacy, resulting in backtracking over assignment. Another important difference is the granularity size of processes. In general, logic programming languages have much smaller tasks than explicitly controlled procedural languages. These attributes result in a significantly higher memory bandwidth requirement than for procedural languages [19]. There are various means of reducing the bandwidth requirement, and certainly both compiler optimization and hardware support are necessary to attain high performance. This paper discusses shared-memory multiprocessor hardware support in the form a cache design [10] optimized for the memory accessing characteristics of parallel logic programming languages.

Several cache protocols have been proposed, e.g., [1, 2, 5, 7, 13, 18], each aiming to solve the cache coherency problem and reduce common bus traffic. The cache protocol described here, called the PIM (Parallel Inference Machine) cache, is similar to the Illinois protocol [13]; however, the PIM protocol has an additional state, distinguishing between SM (shared modified) and S (shared). The new features added for logic programming architectures are the direct write (**DW**), read purge (**RP**), read invalidate (**RI**), and exclusive read (**ER**) commands, described in detail in Section 3.2.

In summary, parallel logic programming languages display different (more severe) memory referencing and process control characteristics than do procedural languages such as C in a multi-user UNIX environment. Considering shared memory multiprocessors, such as the Sequent [15], these different characteristics translate to a general requirement for higher memory bandwidth, an efficient locking mechanism for short intervals, and an efficient communication mechanism necessary for balancing the load of many small tasks.

This paper is organized as follows. Motivation for the PIM cache design is given in Section 2, with a description of the memory referencing characteristics of the parallel KL1 architecture. Section 3 describes the cache protocol. Section 4 presents the results of a performance evaluation of the cache with a software KL1 emulator and cache simulator. Finally conclusions are presented.

2 Motivation for Cache Design

To motivate the coherent cache design presented later in this paper, it is beneficial to review the KL1 language based on FGHC, and its corresponding architecture. KL1 [3] is designed for ICOT's PIM [1] [6] by adding system programming support features to FGHC. Other similar languages and architectures based on the Warren Abstract Machine

*E-Mail: goto%icot.jp@relay.cs.net

[†] Currently at Mitsubishi Electric Corporation, MIEL, 5-1-1 Ofuna Kamakura-city, Kanagawa 247

[‡] Currently at University of Tokyo, RCAST, 4-6-1 Komaba, Meguro-ku, Tokyo 153

[§] Mita Kokusai Bldg 21F, 1-4-28 Mita, Minato-ku, Tokyo 108

[1] Figure 4 shows the PIM pilot machine.

© 1989 ACM 0884-7495/89/0000/0025$01.50

(WAM) [22] are surprisingly similar in their requirements, and can therefore also advantageously utilize the cache optimizations. See Tick [20] for a performance analysis of one such architecture, Aurora [9], on the PIM cache.

2.1 FGHC: Base of KL1

Flat Guarded Horn Clauses (FGHC) [21] is a language based on Horn clauses of the form: H :- $G_1, ...G_m | B_1, ..., B_n$. where H is the head of the clause, G_i are guards, "|" is the commit, and B_j are the body goals. In FGHC, as in Prolog, procedures are composed of sets of clauses with the same name and arity. Unlike Prolog, there are no nondeterminate procedures. Execution proceeds by attempting unification between a goal (the caller) and a clause head (the callee). If unification succeeds, execution of the guard goals are attempted. These goals can only be system-defined builtin procedures, e.g., arithmetic comparison. If the guard succeeds, the procedure call "commits" to that clause, i.e., any other possibly good candidate clauses are dismissed. If the head or guard fails, another candidate clause in the procedure is attempted (if all clauses fail, the program fails). There is a third possibility however: that the call *suspends*. This is described in detail below.

FGHC restricts unification in the head and guard (the "passive part" of the clause) to be input unification only, i.e., bindings are not exported. Output unification can be performed only in the body part (the "active part"). These restrictions allow AND-parallel execution of body goals and even OR-parallel execution of passive parts during a procedure call (the implementation discussed herein executes passive parts sequentially and executes body goals in a depth-first manner). Synchronization between processes is inherently performed by the requirement that no output bindings can be made in the passive part. If a binding is attempted, the call *potentially* suspends. If none of the clauses succeeds, and one or more potentially suspend, then the procedure call suspends (possibly on multiple variables).

When any of the variables to which an export binding was attempted are in fact bound (by another process), the suspended call is resumed. These semantics permit stream AND-parallel execution of the program, i.e., incomplete lists of data can be streamed from one parallel process to another in a producer/consumer relationship. For example, when a stream runs dry, the consumer receives the unbound tail of a list and suspends. When the producer generates more data, the consumer is resumed and continues processing the transmitted data.

2.2 KL1 Architecture

The current KL1 system [14] uses the five main (shared-memory) storage areas: heap, instruction, goal, communication, and suspension. Unlike similar Prolog architectures, the goal and heap areas *cannot* be managed in a stack-like fashion. The goal, suspension and communication areas are managed with free-lists. The heap is allocated from the top, like an ever-growing stack, and can only be reclaimed with a general garbage collection (GC) mechanism.

The KL1 execution model is a reduction mechanism wherein the initial user query (a set of goals) is reduced to the empty set. A goal is represented by a goal record, similar to a Prolog environment [22]. Reducible goal records are stored as a linked list (goal list), one per processing element (PE), and reduction proceeds in a depth-first manner. However, unlike a Prolog environment, KL1 goal arguments consisting of unbound variables and structures are both indirectly stored in the heap.

A single goal is reduced in the following manner [8]. The goal is dequeued from the goal list and its arguments are loaded into a register set. The compiled code sequence corresponding to the goal is executed, attempting to commit to one of the clauses of its procedure. If a clause commits, the body instructions cause body goals to be created and pushed onto the front of the goal list. Otherwise, if all clauses fail, the program fails. Otherwise, if no clause commits, but one or more can suspend, the goal is recreated from registers to the goal area, and the synchronizing variables are hooked to the goal, each via a suspension record. The goal is now floating, i.e., not linked to the goal list. A single variable may be hooked to multiple waiting goals, thus suspension records can be linked. When a hooked variable is bound at some point, a resumption routine is executed which relinks the floating goal(s) to the goal list and reclaims the suspension record(s).

The current KL1 system uses an on-demand scheduler, i.e., idle PEs request a goal from busy PEs [14]. The communication area is used primarily for passing these goal request and reply messages. Message records are only two words and are usually written once and read once. Goals are similar: goal records are always created in the register file. However, when enqueued/dequeued onto/from the goal list, memory is always written once and read once. Suspension has the same effect as enqueue.

When binding a variable in KL1, the variable must be locked to prevent other PEs from concurrently attempting an inconsistent binding. When actively unifying two variables, substructures must be locked in a careful manner to prevent races. Sending messages (to and from idle and working PEs during load balancing) must be also locked.

2.3 Optimizing Cache Performance

As discussed above, each PE in a KL1 system executes goal reductions of a relatively small granularity (compared to procedural languages). Thus there are significant differences in KL1 memory referencing characteristics from the characteristics of conventional multiprocessor systems like the Symmetry [15]. First, the PEs communicate more often with each other, through logical variables, than the usual parallel processing on the Symmetry system, because parallel goals share logical variables. Thus it is important for a locally parallel cache to have an efficient cache-to-cache data transfer mechanism as well as to work as a shared global memory cache. Second, there are many exclusive accesses to communicate through shared logical variables. The frequency of locking shared data during KL1 execution is relatively high. However, we can expect that exclusive memory accesses seldom conflict with each other [14]. Therefore, we require a high-speed lock mechanism that uses hardware support and works efficiently, at least when one lock does not conflict with other locks. Third, because of the single-assignment feature of KL1, data write frequency is higher than in conventional languages [16]. Hence, it is necessary to reduce the number of copyback operations from cache to shared global memory.

In normal write operations, a fetch-on-write strategy is used, i.e., a cache block is allocated in the cache for both read and write misses. As previously stated, in KL1, new data structures are created dynamically in the heap and goal areas. Thus it is not necessary to fetch-on-write when a new cache block is allocated for a new data structure. A PE can write data in a cache block without fetching the contents from shared memory. This is the motivation for the *direct write* command. Similar optimizations have been, for instance, suggested by Tick [19], and implemented in the PSI machine [11].

As stated in Section 2.2, KL1 goal records are referenced with a strict write-once, read-once rule. When KL1 goals are distributed during load balancing from one PE to another, the goal records both in the sender's cache and in the receiver's cache are useless after the receiver PE reads the contents. Thus meaningless swap-in and swap-out can be avoided by invalidating the sender's cache block after cache-to-cache transfer and by purging the receiver's cache block after the receiver finishes reading. This is the motivation of the *exclusive read* command. Exclusive read is used in conjunction with direct write. For example, the sender PE creates a goal record in the goal area with the direct write command (i.e., without fetching the contents from shared global memory). Then the receiver PE reads the goal record with exclusive read. This idea can also be applied to general inter-PE communication via a strict write-once and read-once communication buffer. Although exclusive read must be used carefully so as not to confuse cache coherency, it can reduce common bus traffic by avoiding useless swap-in and swap-out operations.

Lock operations are essential in shared global memory architectures. The KL1 language processor uses lock operations for heap and communication area accesses [14]. The frequency of locking shared data is high; however, actual lock conflicts seldom occur [14]. Therefore, it is effective to introduce a hardware lock mechanism that has less overhead when there are no lock conflicts.

3 PIM Cache Protocol

Copyback cache protocols have been proved effective for reducing common bus traffic in shared-memory multiprocessors for procedural languages, as shown by Goodman [5] and Archibald [1], among others. For logic programming languages, Tick [19] shows that AND-parallel Prolog benefits from copyback even more than procedural languages because of Prolog's high write bandwidth requirement. Thus the basis for the PIM cache is a copyback protocol.

Common shared-memory coherent cache protocols, e.g., [1, 2, 13, 17], use both invalidation and broadcast to ensure all caches are consistent. Both types of protocols have been compared in the literature [1, 4]. Invalidation reduces common bus traffic when the frequency of shared block write accesses is low. Broadcast is better when many PEs frequently write data to the same shared blocks. Considering the single-assignment feature of FGHC, the base language of KL1, most logical variables are shared by only two KL1 goals. In other words, most PE communication has a one-to-one correspondence with regard to heap and goal record accesses. Thus broadcasting is not necessary for most programs, and invalidation suffices.

KL1 write frequency is higher than that of procedural languages, and in addition, cache-to-cache data transfer occurs often. Therefore, when transferring a dirty block on the shared bus, avoiding the update of shared global memory can reduce the busy ratio of memory modules, more so than for a procedural architecture.

3.1 Cache and Lock Directories

The PIM cache has five states: EM (Exclusive modified—The block is exclusive and modified. It is necessary to swap out), EC (Exclusive clean—The block is exclusive and unmodified. It is not necessary to swap out), SM (Shared modified—The block is modified and perhaps shared. It is necessary to swap out), S (Shared—The block is perhaps shared. It is not necessary to swap out), INV (Invalid—The block is invalid). This protocol is similar to the Illinois protocol [13]; however, the PIM protocol has an additional state, distinguishing between SM and S. The SM state is not necessary in the Illinois protocol because modified blocks are always copied-back to shared memory when transferring between PEs, and thus the blocks become unmodified. This reduction in modified blocks likewise reduces swap-out bus traffic. On the other hand, such a protocol will cause the busy ratio of shared-memory modules to increase if the cache-to-cache data transfer rate is relatively high as in the KL1 system. Therefore, the shared modified state (SM) was included in the PIM cache.

In addition to the cache directory, there is a separate lock directory, with three states [2]: LCK (Lock—The address is locked by an **LR** operation of the PE, and other PEs are not waiting to be unlocked), LWAIT (Lock waiter—The address is locked by this PE. In addition, one or more PEs are waiting to be unlocked), EMP (Not locked—empty). When a PE locks an address, the address is registered in the LCK state in the lock directory at first. Subsequently if the cache block is referenced by another PE, the state of the lock directory changes from LCK to LWAIT. In this case, the requester enters the busy wait state, and retries the operation after receiving the unlock (UL) bus command.

Three standard lock operations, **LR** (lock and read), **UW** (write and unlock), and **U** (unlock), are offered. The PIM cache uses busy-wait locking because it usually allows locking and unlocking to occur in zero time [2]. The hardware locks reduce bus traffic in two key ways. **LR** operations do not require bus commands when they hit in exclusive cache blocks (EC or EM). Also, **UW** and **U** operations use the bus only when other PEs are waiting to be unlocked, i.e., in the LWAIT status. In the KL1 architecture, actual lock conflicts seldom occur because the locking periods are, short [14].

The lock and cache directories are separated because if the lock information is held in the cache directory (such as in cache-state locking [2]), there are three major difficulties. First, it is difficult to distinguish every locked word in the same cache block. Second, it is difficult to swap out the block that contains the locked address. Third, it is costly to add lock states for each cache tag. These problems can all be solved by introducing a separate lock directory. The lock directory contains the locked word address and state to enable word-by-word locking. The lock directory controller snoops the common bus to detect and inhibit access to the locked address even if the locked address is swapped out. Therefore, multiple locks in the same cache block can be distinguished. We think only one or two lock entry per directory is needed in most parallel logic programming architectures.

3.2 Memory Operations

Memory operations include read (**R**) and write (**W**) in addition to the following optimizations.

(1) DW (address): Direct write—To avoid swap-in overhead, the **DW** command writes data to an unused memory area without fetching a cache block. The **DW** command acts in two different ways according to the relative position of a cache block, as follows:

(i) When the memory address is on a cache block boundary and a cache miss occurs, a new cache block is allocated and the data is written to the block without fetching the original data from shared global memory. In this case, it must be clear that remote caches do not have a corresponding cache block.

(ii) When the memory address is not a cache block boundary, the cache controller automatically replaces **DW** with **W**.

The **DW** command can be applied locally only when the target cache block entry does not exist in a remote cache. This restriction is necessary to guarantee cache coherency when the corresponding cache block is swapped out. Depending on the precise definition of "cache block boundary," **DW** will work either for upward-growing or downward-growing stacks. To optimize both, two commands are necessary.

(2) ER (address): Exclusive read—This command reads data and invalidates or purges a cache block to avoid swap-out overhead (the word "purge" is used when purging its own cache entry). The **ER** command acts in three different ways according to the relative position of a cache block:

(i) When the target address misses but the block resides on another PE, and the address is not the last word of a block, a cache-to-cache transfer from a supplier PE occurs. In this case, the supplier cache block is invalidated after the cache block transfer. This operation is called read invalidate (**RI**).

(ii) When the target address hits and ***is the last word in a block***, the block in the receiver cache is forcibly purged, after the last word of the block is read. This is the same as read purge (**RP**), described below.

(iii) Otherwise the cache controller automatically replaces **ER** with read **R**.

The **ER** command is used when the contents of a cache block are not required (in cache) after the PE reads the contents (into registers).

(3) RP (address): Read purge—This command, like **ER**, reads data by invalidating or purging a cache block to avoid swap-out overhead. The **RP** command acts in two different ways according to a cache hit or miss, as follows:

(i) When the target address hits, the cache block is forcibly purged after the **RP** operation.

(ii) When the target address misses and the cache block resides on another PE, the supplier cache block is invalidated after the data block transfer, and the fetched cache block is also forcibly purged after the **RP** operation.

The **RP** command is used when a cache block cannot be purged by **ER**, i.e., when the number of words of the reading area is not a multiple of the cache block word size. In this case, the last word of the reading area is read by **RP**.

(4) RI (address): Read invalidate—This command is effective for avoiding invalidate bus commands when the data is rewritten just after it is read from other PE cache.

3.3 Specifications of Bus Commands and Responses

There are three bus commands and one response to implement the locally parallel cache mechanism for the PIM. Additional bus commands or responses are not necessary for the optimized memory operations.

(1) F (address): Fetch—The fetch command is a request to fetch a cache block from other PEs or shared global memory.

(2) FI (address): Fetch and invalidate—The fetch and invalidate command is a request to fetch a cache block from other PEs or shared global memory, and to invalidate the cache blocks of all other PEs, including the supplier PE of a cache-to-cache transfer.

(3) I (address): Invalidate—The invalidate command is a request to invalidate the cache blocks of all other PEs.

(4) H: Hit—response for **F** and **FI** requests.

Two additional bus commands and one response are necessary for lock operations.

(1) **LK (address):** Lock—This is a broadcast message to report that a specified address will be locked. The **LK** bus command is always used together with the **FI** or **I** bus commands. Note that the **LK** bus command is used only when the **LR** memory operation misses or hits to shared blocks.

(2) **UL (address):** Unlock—This is a broadcast message that a specified address has been unlocked, in the LWAIT state. If the locked address is not in the LWAIT state, in other words, another PE does not refer to the address, then the **UL** bus command is not broadcast. This is an optimization to reduce the common bus traffic.

(3) **LH:** Lock hit—This is a response to the **F**, **FI**, or **LK** bus commands. The **LH** response shows that the referred address is locked. The requester PE which received the **LH** response starts busy waiting. The requester PE retries a memory reference after it receives the **UL** bus command. The common bus is not used during busy waiting cycles [2].

In summary, the PIM cache protocol is a copyback locally parallel cache with invalidation (of other caches when writing to shared blocks) and no copyback to shared memory (during a cache-to-cache transfer). Optimizations include the direct write, exclusive read, read invalidate, and read purge memory commands. A busy wait lock mechanism with a separate lock directory is implemented. Refer to Matsumoto [10] for the complete state transition tables of the PIM cache protocol.

4 Cache Performance Evaluation

The cache design was evaluated with a parallel simulator written by A. Matsumoto. The simulator executes in cooperation with an abstract machine emulator—in this paper, we only discuss the parallel KL1 emulator written by M. Sato. These tools currently run on the Sequent Symmetry multiprocessor (refer to Tick [20] for a detailed description of these tools). Each PE runs a reduction engine for the abstract machine, dynamically feeding memory requests to a local cache simulator. The cache simulators artificially synchronize among themselves at each simulated bus request. This synchronization retains the accuracy of the model's parallelism without overly impacting simulation speed. These tools represent an improvement in accuracy over previous studies [10] which used a pseudo-parallel emulator, synchronizing only on each reduction.

Modeling a real architecture on a target host, with an emulated architecture on a partially mapped host, requires creating a correspondence between emulator variables and target machine registers and memory. In the measurements presented here, we assume a very liberal correspondence of architecture state to registers. Most emulator variables are considered either not necessary for the target architecture, or able to be allocated to temporary registers. In addition, the state and argument registers of the architectures, based on the WAM [22], are also mapped onto registers. For KL1, this means that references to goal queue pointers, processor status, communication buffer pointers, interrupt status, suspension stack pointers, meta-counts, and GC pointers are *not* counted as memory references. This is of course a best case assumption. Memory references to the major storage areas of the architecture (for KL1: heap, goal, instruction, suspension, and communication) are instrumented as target architecture memory references. Note that the system measured uses stop-and-copy GC, and that inclusion of incremental GC will significantly affect heap referencing characteristics [12].

bench	**lines**	**sec.**	**su**	**reduct**	**susp**	**instr**	**ref**
Tri	182	49.3	5.8	666233	1	13.0M	28.9M
Semi	104	87.5	4.8	268820	23487	4.8M	23.1M
Puzzle	151	55.3	6.5	849539	3069	15.6M	29.1M
Pascal	310	16.6	6.1	302432	17681	5.0M	10.5M

Table 1: Short Summary of Benchmarks on Eight PEs

Mem Ref	**inst**	**data**	**heap**	**goal**	**susp**	**comm**
$E(inst + data)$	42.87	57.13	34.31	20.71	0.26	1.86
$\sigma(inst + data)$	13.17	13.17	23.86	11.82	0.37	1.23
$E(data)$	—	—	60.06	36.25	0.46	3.26
Bus Cyc.	**inst**	**data**	**heap**	**goal**	**susp**	**comm**
$E(inst + data)$	4.52	95.48	65.70	11.16	1.14	17.49
$\sigma(inst + data)$	3.45	3.45	15.74	7.15	1.22	9.20
$E(data)$	—	—	68.81	11.69	1.19	18.31
Tri	7.15	92.85	43.00	22.68	0.00	27.16
Semi	0.93	99.07	79.66	4.80	1.17	13.45
Puzzle	8.69	91.31	81.10	5.59	0.26	4.36
Pascal	1.30	98.70	59.03	11.57	3.12	24.98

Table 2: % Memory References and Bus Cycles by Area

In the following sections, the performance of the PIM cache design is evaluated with respect to the KL1 architecture, the target architecture for which the cache was designed.

4.1 Characteristics of the Benchmarks

Measurements of four KL1 benchmarks (written in pure FGHC) are analyzed in this paper. High-level characteristics of the benchmarks are given in Table 1 (for discussion and code listings of these programs, see Tick [20]). Lines of static code, execution time (sec.) and relative speedup (su) on eight PEs, reductions, suspensions, (millions of) KL1 instructions executed, and (millions of) emulated memory references (both instruction and data), are given. For some benchmarks, these high-level statistics are sensitive to emulator timing, but the accuracy is sufficient for our needs.

Table 2 shows the memory access and bus traffic characteristics, averaged over the benchmarks. Means (E) and standard deviations (σ) for instructions plus data and data only are calculated. The simulation assumed an eight cycle memory access, a one word bus, and eight PEs, each with a four-word block, four-way set-associative, four Kword I+D cache (with *no* optimized commands). These statistics are useful to gain insight into the relative bandwidth requirements of the different storage areas.

Table 2 shows 43% of the memory references are used for fetching instructions. In subsequent sections, the various

operation	R	LR	W	UW+U
$E(inst + data)$	78.95	2.66	15.71	2.70
$\sigma(inst + data)$	8.01	0.82	6.57	0.91
$E(data)$	58.91	5.14	30.73	5.22
$\sigma(data)$	18.61	2.23	14.47	2.38
$E(heap)$	57.64	10.39	21.38	10.60
$\sigma(heap)$	21.22	5.26	11.35	5.48
Tri	54.62	12.06	21.27	12.06
Semi	93.17	1.70	3.42	1.71
Puzzle	41.88	11.90	34.26	11.96
Pascal	40.87	15.88	26.57	16.68

Table 3: Percentage of Memory References by Operation

cache parameters are examined in more detail. Here we see that 95% of all bus cycles are devoted to data indicating that the cache is very successful at reducing the instruction bandwidth requirement. Larger benchmarks are anticipated to offer less instruction locality and therefore generate a larger percentage of instruction bus traffic. In general however, it is most important to reduce the data bandwidth requirement further with cache optimizations.

Heap access frequency is 34%, yet the heap accounts for 66% of all bus cycles, significantly greater than the other areas. The dynamic heap area size is very large (e.g., over 80% of all shared memory for the **BUP** benchmark as reported in [10]) and access locality low. The communication and goal area management, based on free-lists, helps to reclaim space and retain locality; however, still these areas account for 29% of all bus cycles. Note that the shared communication area is particularly troublesome: less than 2% of all memory requests require more than 17% of all bus cycles.

Table 3 shows the memory references by operations. Data write frequency is 36% (**W** + **UW**), somewhat lower than 47% for Prolog [19]. This statistic is highly variant however: **Semi** with only 7% data writes lowers the average. Locking (**LR**) and unlocking (**U**, **UW**) frequency is more than 5%. Examining heap accesses in more detail, we find that for these benchmarks, heap write frequency varies from 5–46%. Heap lock/unlock reference frequency (**LR** + **UW** + **U**) varies from 3–33%. The high variance is again due to **Semi**. In any case, these statistics show that logic programs generate significant heap write traffic because of dynamic structure creation. In addition, dependent AND-parallel programs generate significant heap lock/unlock traffic to protect bindings. The cache design introduced helps mollify and alleviate much of these overheads (see Section 4.7).

4.2 Cache Simulation Parameters

There are many complex tradeoffs made in cache design. We choose to concentrate, in this paper, on the reduction of common bus traffic, which is of critical importance in the design of tightly-coupled shared-memory multiprocessors. In the following sections, we discuss the organization of relatively small caches (16K words and less), always with bus bandwidth reduction as our primary figure of merit. Unless otherwise stated, the simulations were run for eight PEs, where each PE's cache memory is four Kwords, four-way set-associative with 256 columns and four-word blocks. Perturbations of this base model are examined in subsequent sections. The simulator models a common bus used for swap-in from shared memory, swap-out to shared memory, cache-to-cache transfer between PEs, and invalidation. Additional assumptions are:

(1) The width of the common bus is one word, which consists of tag and data parts. Separate address and data buses are not distinguished. Therefore, it is assumed that an address cannot be sent with data during the same cycle.

(2) It takes eight cycles to access shared memory. However, a swap-out write operation to shared memory is hidden by a subsequent memory operation.

(3) The common bus is not freed until one memory operation is completed.

Given the above assumptions, there are six common bus access patterns: swap-in from shared memory with swap-out (13 cycles), swap-in from shared memory without swap-out (13 cycles), cache-to-cache transfer with swap-out (10 cycles), cache-to-cache transfer without swap-out (7 cycles), swap-out only (5 cycles—this access pattern appears only in **DW**), invalidation of other PEs' cache blocks (2 cycles).

This model produces a raw bus cycle count. We refrain from introducing a ratio statistic (as in [10, 20]) to avoid confusion. When the above bus width and memory access time assumptions are modified, the bus access cycle times change as does the raw cycle count. Experiments [20] indicate that bus traffic is insensitive to memory access time because most bus traffic is cache-to-cache. Increasing bus width, however, more significantly decreases bus traffic, as shown in Section 4.4.

4.3 Effects of Cache Block Size

Figure 1 shows the relationship between cache block size (in words), miss ratio and bus traffic, for four-way set-associative, four Kword I+D caches (with all optimized commands). Whereas miss ratio improves significantly with increasing block size, the difference in bus traffic for two and four word blocks is relatively small. Above four words, bus traffic is restrictive. Since two-word blocks require about twice the cache address array size as four-word blocks, the latter is most suitable for the PIM cache. Note that in sequential Prolog studies, Tick [19] also found four-word blocks to be optimal. Essentially, logic programming languages, without arrays and with more procedure calls (and suspensions, failures, etc.) than procedural languages, can make less efficient use of large block sizes because there is less spatial locality. Matsumoto [10] found that two-way set-associative PIM caches produce 18% more bus traffic than four-way (for the **BUP** benchmark), whereas direct-mapped caches create significantly greater bus traffic.

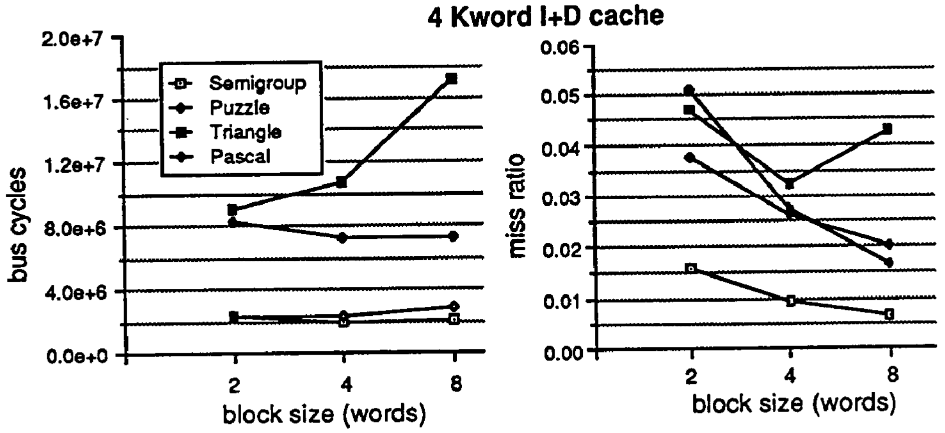

Figure 1: Cache Block Size vs. Cache Miss Ratio and Bus Traffic

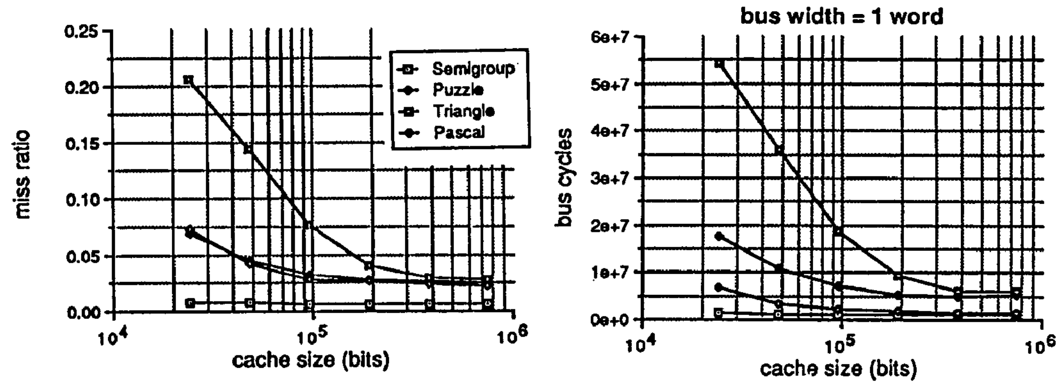

Figure 2: Cache Capacity vs. Bus Traffic

4.4 Effects of Cache Capacity

Figure 2 shows the relationship between cache size, miss ratio and bus traffic, for four-word block I+D caches (with all optimized commands) of sizes 512–16K data words. The plots assume a 5 byte data word and account for directory size, e.g., a "four-Kword cache" is 190000 bits. In **BUP**, if the block size is increased above eight words, bus traffic increases in spite of the increased capacity of the data array [10], thus we limit our measurements to four-word blocks. The knee of both the miss ratio and bus traffic curves is at 4 x 10^5 bits (8 Kword data cache). **Semi** is seen to have a small working-set that is captured in even the smallest cache. **Puzzle** generates a constant 5 x 10^6 more bus cycles than **Pascal**, although they both achieve the same miss ratio. **Puzzle**, with more touched code and larger data structures than **Pascal**, generates a great deal of swap in/out and cache-to-cache traffic to achieve an equal miss ratio. **Tri** has the most touched code of the benchmarks, causing poor code locality on small caches and thus excessive bus traffic. In fact, much of the I+D cache benefit is due to instruction bandwidth reduction. Analyzing data-only caches (not shown), bus traffic is about twice that of I+D caches for the largest caches. Considering a two-word bus width, the benchmarks exhibit a decrease in traffic, to 62–75% of the one-word bus traffic (both assuming an eight cycle memory access and non-overlapped bus). This percentage decrease is insensitive to the benchmark and increases slightly with decreasing cache size.

4.5 Effects of the Number of PEs

Figure 3 shows the relationship between the number of PEs and bus traffic. **Tri** creates a search tree of height 12 with a branch factor of 36 at each node. The inability of the simple KL1 scheduler to balance this load causes excessive bus traffic due to task distribution. **Pascal** shows similar, but less radical, characteristics. Analyzing the separate areas contributing to bus traffic, we find that by percentage of total bus traffic, communication increases from 0–29% and suspension increases from 0.8–2.1% (average from all benchmarks) when increasing from one to eight PEs. At the same time, heap bus traffic decreases from 71–45%, and other areas remain approximately the same. Thus inter-PE communication becomes a dominant factor in parallel processing for more than four or eight PEs. It is likely that about eight high-performance PEs will be connected for one common bus of current specifications in the PIM [10, 6].

4.6 Effects of Cache Optimizations

The effects of the cache optimizations at reducing the bus bandwidth requirement are summarized in this section. Recall from Section 3.2 that direct write (**DW**) reduces swap-in overhead, exclusive read (**ER**) and read purge (**RP**) (in con-

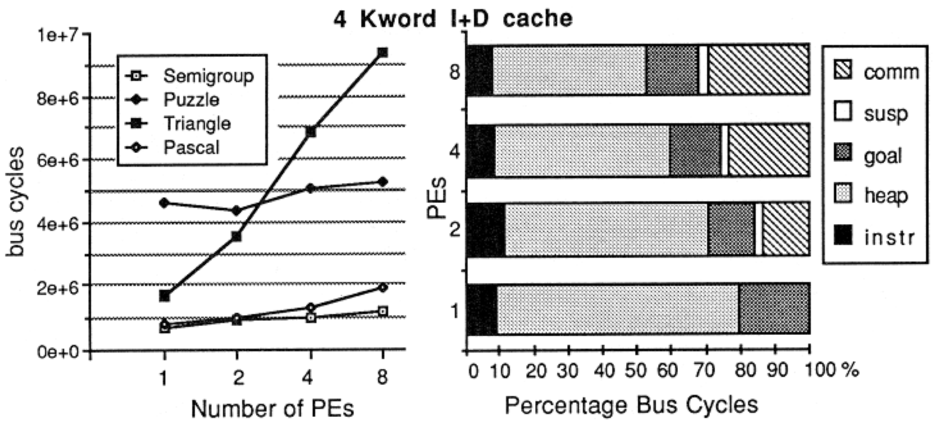

Figure 3: Number of PEs vs. Bus Traffic

junction with **DW**) reduce swap-out overhead, and read invalidate (**RI**) reduces bus invalidations (**I**). Table 4 shows, for each benchmark, the number of bus cycles relative to a non-optimized cache ("None"), for several different optimizations (in each column). The "Heap" optimization is **DW** used only in the heap area. "Goal" is **ER**, **RP** and **DW** used only in goal area. "Comm" is **RI** used only in the communication area. "All" uses all optimizations. As can be seen, **DW** contributes almost all of the savings, with the other optimizations most effective for **Tri**.

More precisely, the **DW** commands for the heap area reduce the swap-in from global shared memory to 10% in **Tri** and 55% in **Puzzle**. The **DW** commands are very effective not only for reducing bus traffic but also for avoiding the CPU waits for fetching from memory. The **ER**, **RP** and **DW** commands for the goal area decrease meaningless swap-out by about 2–10%. The **RI** commands for the communication area can avoid about 60–70% of invalidate (**I**) bus commands. As shown in Figure 3, **Tri** bus traffic increases with the number of PEs because load balancing of many small tasks is a dominant factor. The **ER**, **RP** and **RI** commands are effective in reducing bus traffic for precisely this situation: when many goals are distributed within PEs for load balancing and when the inter-PE communication is a dominant factor of the bus traffic. The **RI** commands avoid only unnecessary **I** bus commands. Thus **RI**, while not reducing the bus traffic in Table 4, becomes more effective in reducing bus traffic as bus width increases or memory access time decreases.

4.7 Effects of Lock Protocol

As stated in Section 3, the lock mechanism reduces the bandwidth requirement because the **LR** operation does not require bus commands when it hits in an exclusive cache block (Ec or EM), and the **UW** and **U** operations use the bus only when other PEs are waiting to be unlocked, which is rare. Table 5 shows the hit ratios of the **LR** operations in general, **LR** directed to exclusive cache blocks, and **UW** and **U** operations directed to non-waiting locks (LCK directory state). As shown, the proposed lock protocol avoids most of the bus traffic for these lock/unlock operations.

	bus cycles relative to no-opt.				
benchmark	**None**	**Heap**	**Goal**	**Comm**	**All**
Tri	1.00	0.62	0.80	0.83	0.52
Semi	1.00	0.65	1.00	0.99	0.62
Puzzle	1.00	0.55	0.98	0.98	0.51
Pascal	1.00	0.64	0.94	0.96	0.60

Table 4: Effect of Optimized Cache Commands in Reducing Bus Traffic

	Tri	**Semi**	**Puzzle**	**Pascal**
LR hit-ratio	0.743	0.912	0.959	0.847
LR hit-to-Exclusive	0.658	0.910	0.954	0.816
U, UW hit-to-No-waiter	0.999	0.993	0.997	0.976

Table 5: Hit Ratios of No Cost Lock Operations

5 Conclusions

This paper describes the design and estimated performance of a coherent cache for parallel logic programming architectures. The cache is optimized around the KL1 execution model; however, it is general enough to execute other architectures. Memory access characteristics of KL1 benchmarks, gathered by emulation, indicate that data write frequency is 36%. This value is slightly lower than Prolog (because KL1 does not backtrack), but higher than in conventional languages. Therefore the PIM cache is based on a copyback protocol. In normal write operations, a fetch-on-write strategy is used. However, in KL1, and other WAM-based architectures, new data structures are created dynamically on the top of the heap area. Therefore, it is not necessary to fetch the contents of shared global memory when a cache block is allocated for a new data structure. In addition, writing goal records also need not fetch their contents. To accomplish this, the *direct write* command was introduced.

In KL1 and other parallel architectures, PE communication (for goal distribution, etc.) uses a shared message buffer. In this case, swap-in and swap-out of meaningless data can be avoided by invalidating the sender's cache block after a

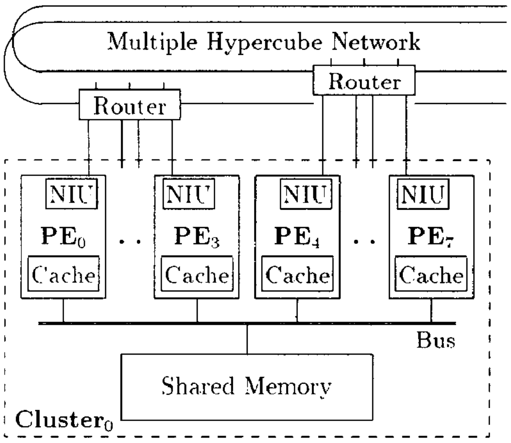

Figure 4: The PIM Pilot Machine: PIM/p (part)

cache-to-cache transfer and by purging the receiver's cache block after the receiver finishes reading. To accomplish this, the *exclusive read* and *read purge* commands were introduced.

These new commands can reduce common bus traffic by avoiding useless swap-in and swap-out operations. Cache simulations indicate that these optimizations reduce bus traffic by 40–50% with respect to an unoptimized system. Direct write affords 35–45% reduction and other optimizations only 5% reduction. From our preliminary data on the Aurora system [20], we believe these optimizations will prove effective on other parallel logic programming architectures as well.

The PIM cache three-state lock protocol was shown to be effective at reducing the bus traffic of lock/unlock operations: for KL1, no bus cycles are needed for the high percentage of lock reads hitting in exclusive blocks and unlocks to non-waiting locks. Locking efficiency aside, we feel however that the most critical bottleneck of parallel logic programming architectures is the high communication cost of load balancing. We have illustrated this with the KL1 **Tri** benchmark, but the problem extends to non-committed choice architectures as well [20].

A version of the PIM cache described here is currently being implemented for ICOT's Parallel Inference Machine (PIM). See Figure 4, and Goto [6] for the PIM architecture overview.

Acknowledgements

We wish to thank Mr. M. Sato for developing the parallel KL1 emulator. We also wish to thank all research members of the PIM R&D project for their fruitful discussions. Finally, we would like to thank ICOT Director, Dr. K. Fuchi, and the chief of the fourth research section, Dr. S. Uchida, for their valuable suggestions and guidance. E. Tick was supported by NSF Grant No. IRI-8704576.

References

[1] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions of Computer Systems*, 4(4):273–298, 1986.

[2] P. Bitar and A. M. Despain. Multiprocessor Cache Synchronization. In *13th ISCA*, pages 424–433, June 1986.

[3] T. Chikayama et. al. Overview of the Parallel Inference Machine Operating System PIMOS. In *Int. Conf. on 5th Gen. Comp. Sys.*, Tokyo, November 1988.

[4] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *15th ISCA*, pages 373–382, June 1988.

[5] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *10th ISCA*, pages 124–131, 1983.

[6] A. Goto et. al. Overview of the Parallel Inference Machine Architecture PIM. In *Int. Conf. on 5th Gen. Comp. Sys.*, Tokyo, November 1988.

[7] R. H. Katz et. al. Implementing a Cache Consistency Protocol. In *12th ISCA*, pages 276–283, June 1985.

[8] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Int. Symp. on Logic Prog.*, pages 468–477, August 1987.

[9] E. Lusk et. al. The Aurora Or-Parallel Prolog System. In *Int. Conf. on 5th Gen. Comp. Sys.*, Tokyo, November 1988.

[10] A. Matsumoto et. al. Locally Parallel Cache Design Based on KL1 Memory Access Characteristics. Technical Report 327, ICOT, 1987.

[11] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Int. Symp. on Logic Prog.*, pages 104–113, August 1987.

[12] K. Nishida et. al. Evaluation of the Effect of Incremental Garbage Collection by MRB on FGHC Parallel Execution Performance. Technical Report 394, ICOT, 1988.

[13] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *11th ISCA*, pages 348–354, 1984.

[14] M. Sato et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *4th Int. Conf. on Logic Prog.*, pages 338–355. MIT Press, May 1987.

[15] Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

[16] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[17] L.C. Stewart et. al. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8), August 1988.

[18] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *13th ISCA*, pages 414–423, June 1986.

[19] E. Tick. Data Buffer Performance for Sequential Prolog Architectures. In *15th ISCA*, May 1988.

[20] E. Tick. Performance of Parallel Logic Programming Architectures. Technical Report TR-421, ICOT, September 1988.

[21] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[22] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.