CFlex, a Programming Language for the FlexRAM Intelligent Memory Architecture

Basilio B. Fraguela* Jose Renau† Paul Feautrier‡ David Padua†
Josep Torrellas†

* *Dept. de Electrónica e Sistemas, Universidade da Coruña, E-15071, A Coruña, Spain.*
basilio@udc.es
† *Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 61801.*
{renau,padua,torrella}@uiuc.edu
‡ *Laboratoire PRiSM, Université de Versailles St-Quentin, F-78035 Versailles Cedex, France.*
paul.feautrier@prism.uvsq.fr

Technical Report UIUCDCS-R-2002-2287, Dept. of Computer Science, UIUC, July 2002

Abstract

Advances in VLSI technology have enabled the implementation of computer architectures based on Processing-in-Memory (PIM) chips. These architectures can help bridge the growing gap between processor and main memory speeds and, as a result, improve the performance of both numeric and symbolic codes. While PIM organization has been extensively studied from the machine organization point of view, the problem of effectively programming this kind of systems has been largely ignored. In this paper, we address how to program FlexRAM, a general-purpose computer with PIM memory modules. We present CFlex, a family of compiler directives inspired in OpenMP, which enable the development of portable programs. We also explore enhancing the programmability with libraries of functions called Intelligent Memory Operations (IMOs). These libraries are optimized to make use of the PIMs but hide all their complexity. The geometric mean of the speedups achieved for our test suite by a system with one FlexRAM chip programmed with CFlex relative to a conventional machine is 14.4.

1 Introduction

The performance of modern computers with high-frequency processors is often limited by the long latencies and low bandwidths of their memory systems. A promising approach to remove this memory bottleneck is to migrate all or part of the computation to processors embedded in the memory chips. The integration of processors and memory in the same chip, commonly known as Processing-in-Memory (PIM), has been made possible by advances in VLSI technology. In fact, recent advances allow the integration of processors with clock rates matching those found in logic-only chips with DRAM that is only about 10% less dense than what is typical in memory-only chips [1, 2].

Several research groups are studying this new technology following one of three main approaches. In the first approach, the PIM chip contains the main processor [3, 4], as much memory as possible and, in some cases, one or more special-purpose processors such as vector processors. The second approach is to design special-purpose processors or co-processors integrated with much memory

that are oriented to specific tasks [5, 6]. The third approach consists in using the PIM chips as intelligent memories that replace all or some of the standard memory modules of a server or workstation [7, 8, 9]. These latter architectures are able to run in the PIM modules the most memory-intensive parts of the application, which are those for which the current systems exhibit the worst performance.

While much effort has been devoted to the analysis and evaluation of PIM architectures, programming issues have been largely ignored. These issues are particularly important for the third group of PIM architectures, as they potentially include a large number of simple, general-purpose processors that must act in coordination with the main processor(s) of the system.

To address this problem, this paper presents a programming language and its associated runtime system for FlexRAM [7], which is one of these architectures that use PIM chips as intelligent memories. In FlexRAM, the processors in the PIM chips lack most of the synchronization and communication mechanisms found in many traditional multiprocessor systems. This is due to several reasons. First, the PIM chips replace conventional memory chips, which are never masters of the memory bus. As a result, their simple processors can never invoke the main processor(s). Another reason is that the caches in the memory processors cannot be kept coherent with those of the main processor(s) using hardware mechanisms because the system lacks the required connections. Other limitations come from the need to keep the hardware of the memory processors as simple as possible. For example, the memory processors do not have support for atomic operations that would allow them to use well-known synchronization mechanisms based on regular locks and barriers. Overall, all these limitations provide a significant challenge to effectively program the FlexRAM architecture.

The rest of this paper is organized as follows. In the next section, we describe the second version of the FlexRAM system introduced in [7]. Then, Sect. 3 presents the CFlex compiler directives that we propose to program this system. CFlex directives are as simple and easy to use as OpenMP [10] directives, but they are much more flexible. In fact, as the examples in Sect. 4 will show, CFlex directives allow parallelization without extensive reprogramming of more codes in our test suite than standard OpenMP. The operating system extensions and runtime system underlying CFlex are discussed in Sect. 5. In this section, we also discuss the design of libraries that encapsulate the use of PIM processors. These libraries, called Intelligent Memory Operations (IMOs), enhance programmability while providing near optimal performance. The results of an evaluation of FlexRAM when programmed using CFlex are reported in Sect. 6. Finally, Sect. 7 presents conclusions and future work.

2 FlexRAM Architecture

A FlexRAM system is an off-the-shelf workstation or server where some of the DRAM chips in the main memory have been replaced by FlexRAM PIM chips [7]. Each FlexRAM PIM chip contains memory plus many simple, general-purpose processing elements called *PArrays*. The resulting memory system is a versatile accelerator where many processors can access memory with high bandwidth and low latency. Moreover, legacy codes can run unmodified, since they see the FlexRAM chips as standard memory chips.

In our current design, each FlexRAM chip has 64 Mbytes of memory organized in 64 1-Mbyte banks. Each bank is associated to one PArray. Since each PArray can be programmed independently, PArrays can run in SPMD or MIMD mode. Due to density and power restrictions, PArrays are much simpler than the main processor of the system, which we call *PHost*. However, FlexRAM chips do not require any change to the existing memory system specifications to be integrated.

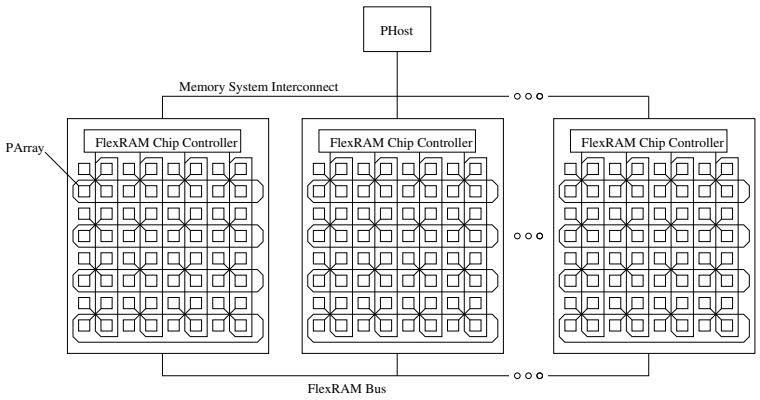

Figure 1: Workstation or server with FlexRAM chips.

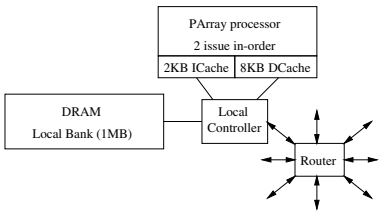

Figure 2: Structure of a PArray and a memory bank inside a FlexRAM chip.

The only requirement is a memory standard that provides additional power and ground signals to enable on-chip processing. A standard that meets these requirements is Rambus [11].

Compared to the original FlexRAM architecture proposed in [7], three main architectural improvements have been made. First, each PArray now has a 8 KByte write-back data cache and a 2 KByte instruction cache. Second, each FlexRAM chip now includes an internal 2-D torus connecting the PArrays, since there are enough metal layers to allow its implementation. The torus enables a PArray to access any of the banks in its chip. The torus is connected to a bus that connects all the FlexRAM chips (FlexRAM bus). Consequently, a PArray can also access the banks in the other FlexRAM chips. Finally, given the high connectivity of PArrays, we remove per-chip controller (PMem) proposed in [7]. The main purpose of PMem was to shuffle data between the PArrays banks. Instead, we only have a much simpler FlexRAM chip controller (*FXCC*) that acts as the interface between the PHost and the PArrays. FXCC provides services for PArray synchronization. Specifically, it contains registers to store the requests from the PHost to the PArrays, such as tasks to spawn, and from the PArrays to the PHost, such as page faults. Figs. 1 and 2 depict the current FlexRAM system.

These architectural changes cause changes in other areas like address translation. Specifically, recall that PArrays use virtual addresses in the virtual space of the application run by the PHost. Each PArray has a relatively small TLB that contains some entries from the PHost's page table. Given the higher connectivity of the PArrays, PArrays can now serve their own TLB misses by accessing the PHost's page table. However, page faults, page migrations, or other operations that involve changes in the page tables must still be managed by the PHost operating system.

2.1 Interprocessor Communication and Synchronization

PArrays communicate with the PHost(s) and with other PArrays in two ways: through memory or via registers in the FXCCs. The input and output data of the tasks executed by the PArrays is typically retrieved from and stored to memory, respectively. However, the commands and service requests that modify the state of the PArrays are stored in registers in the FXCCs. FXCCs are also used for the synchronization among PArrays and between PArrays and PHost when locks are involved. In this section, we examine these communications in detail.

2.1.1 Communication from PHost to PArray

The PHost communicates with the PArrays to spawn tasks on them, answer service petitions, and order maintenance operations such as flushing certain pages from their caches and invalidating the TLB entry associated with those pages. In either case, the PHost issues a command that consists of a command type code and at most two words. For example, assume that the input data of a task are stored in memory. Then, the command to order a PArray to start executing the task includes the address of the routine to execute and a pointer to the input data buffer. The PHost delivers such a command by programming the FXCC of the corresponding FlexRAM chip using special features and reserved code words from the Rambus definition. The FXCC can be envisioned as a memory mapped I/O device with a series of ports that can be written to send control and data signals, or read to check the status of the device. The FXCC temporarily stores the PHost commands and sends them to the corresponding PArray by means of the internal torus in the chip.

2.1.2 Communication from PArray to PHost

When a PArray requires a service from the PHost, such as handling a page fault, it sends a message with the service code and its parameters to its FXCC through the torus. The FXCC cannot deliver this message to the PHost, since only the PHost(s) can be master(s) of the memory bus. Consequently, the PHost periodically polls the FXCCs of the chips to check for potential requests from the PArrays. The FXCCs have registers to hold PArray requests until the next PHost poll.

2.1.3 Synchronization

There are a number of well-known mechanisms for processor synchronization. Unfortunately, many of these mechanisms rely on atomic operations and hardware coherence, which are too expensive to implement in our simple PArrays. Still, FlexRAM chips have a simple yet effective synchronization mechanism based on locks managed by the FXCCs. Such locks may be acquired and released by both PArrays and PHost(s), thus enabling any synchronization pattern in the system. FXCC locks are implemented as tokens. Each FXCC is responsible for a disjoint set of such locks. For those locks that are currently acquired, the home FXCC records the owner processor in a set of registers. Both acquire and release operations are implemented as requests to the home FXCC, which will check its registers to grant or deny the corresponding operation. When an acquire request for a free lock arrives at a FXCC that has run out of free lock registers, it denies the operation until there is a register available to store the owner of the lock.

2.1.4 Data coherence

Our system lacks hardware support for cache coherence between PHost(s) and PArrays. The cheapest and simplest solution to still maintain cache coherence involves at least the ability to flush and invalidate caches. Note that, in this environment, a (total or partial) flush operation of

the write-back caches is always required to ensure that the latest version of the data is in memory and, therefore, visible to other processors.

In our system, we use the following procedure. The PHost flushes its caches before starting a task on the FlexRAM chips. This is done to make sure that the PArrays see the latest versions of data. Moreover, before the PHost executes code that may use results from the PArrays, the PHost invalidates a superset of such variables from its caches. As for the PArrays, when they complete a task, they flush and invalidate their small caches. This ensures that the data generated by the task is visible to the whole system and that, when the PArray resumes, its cache is free of stale variables.

While coherence among the PArrays could be supported with a directory-based hardware scheme, we feel that it is too costly for a system like FlexRAM. Consequently, we choose to restrict our programming scope to problems in which different PArrays write on different data items. The PArray caches include a dirty bit per word so that when a line is displaced from a cache, only the modified words are written back. This way, data coherence is maintained even in the presence of false sharing.

Finally, in applications where several PArrays need to write-share data, these mechanisms are not enough. In these cases, the compiler or the programmer marks as critical sections those code portions where such data is manipulated. The compiler then inserts the corresponding cache flushes, invalidations, and FXCC lock operations to ensure program correctness.

3 CFlex

We have chosen to program FlexRAM adding compiler directives to the sequential version of the programs rather than adding new features to standard programming languages. The reason is that in this way the same program can also be compiled for conventional execution by ignoring the directives, and therefore a single version of the program can be executed both on conventional systems and systems extended with FlexRAM chips.

The translation of CFlex directives has been implemented using the SUIF compiler [12] augmented with a preprocessor, and is currently aimed at C programs, although it can be easily extended to other languages. The basic structure of a CFlex directive is

#pragma FlexRAM *directive-type [clauses]*

There are three classes of directives:

- *Execution modifiers* which indicate how a given piece of the code should be executed. These are the most widely used directives. They include the requests to spawn a given portion of the program for execution on a given processor or group of processors.
- *Data modifiers* to request that data structures fulfill certain conditions. An example are directives to pad one of the dimensions of an array to make their size a multiple of the page size.
- *Executable directives* that are just instructions in the program. Examples of this class include barriers, prefetch operations, etc.

Let us now see in detail the directives proposed and implemented.

3.1 Execution Modifiers

3.1.1 Spawning Tasks

The only directives that are required to use FlexRAM are those ones in this class that define, spawn and synchronize tasks. These directives can only appear in the code executed by a PHost, as the PArrays cannot perform this type of operations. The *directive-type* is the class of processor on which the code will execute and its possible values are `phost` and `parray`. The code of the task to be spawned is the (possibly compound) statement following the directive.

The optional clauses for this directive are:

- `on_home(`*x*`)` to specify that the task should be executed on the PArray on whose memory bank is located the data structure *x*.
- `sync`/`async` to specify whether the task spawning the new task must stop until the new task finishes (`sync`) or it can continue (`async`). The default is tt sync. A task does not finish until all the tasks it has spawned have finished. Asynchronous tasks must be created inside the syntactical scope of a synchronous task, so that there is a point in the program execution flow where they are known to have finished. Such point is the end of the synchronous task inside which they have been spawn. The power, flexibility and simplicity of this approach will be illustrated in Sect. 4
- `if(`*cond*`)` controls the execution of the directive where it appears. The directive is executed only if *cond* is true.
- `else` also controls the execution of a directive. The directive is executed if the *cond* in the `if` clause in the immediately preceding directive is false.
- `shared`, `private`, `lastprivate`, `firstprivate`, `reduction`, `migrate` which have the same semantics as the directives of the same name in OpenMP. In contrast to OpenMP, CFlex allows their application to only one part of a structure (e.g., a field of a struct, an element of a vector). A `migrate` clause has been considered. It designates shared data structures whose pages should migrate to the first PArray that touches them once the task(s) created by this directive begin their execution. This clause has not been implemented for two reasons. First, the high cost of automatic page migration offsets very often its advantages even in architectures, such as the NUMA machines [13], that are much better suited for it than the PIMs. Second, although its functionality is more restricted, our `on_home` clause already allows the migration of computations to the processor that owns certain data at much cheaper cost.
- `flush` specifies which pieces of data need to be flushed from the PHost cache so that the PArrays executing the task(s) have access to the latest version of the data they require. By default the compiler flushes the whole cache of the PHost to ensure that the PArrays do not use outdated data residing in memory. The programmer can use this clause to improve the performance by restricting the flush to a certain set of variables. The special value 0 (`flush(0)`) tells the compiler that there is no need to flush anything excepting the package of input data to the task built from the data scope specifications.
- `pfor` breaks the iterations of a parallel for loop into tasks and inserts a synchronization point for them right after the loop. Its usage will be illustrated in Sect. 4.
- `among` allows to specify among how many tasks do we want to distribute the execution of a parallel loop. By default the compiler will try to split it among all the PArrays

- `initval(`v_1`,`val_1`,`v_2`,`val_2`,...)` inserts code at the beginning of the defined tasks that assigns value val_i to variable v_i. This allows an additional optimization in certain cases by replacing the pass of these values to the task spawned by their creation inside the task.

Notice that CFlex can be used to express parallelism in systems without PIMs because it can generate and synchronize parallel tasks at the PHost level without ever having to refer to PArrays. In fact, CFlex is an easy and sensible approach to port existing parallel codes to NUMA systems thanks to clauses like `on_home` or `migrate`. As an additional benefit, the `sync`/`async` clauses enable us to generate new tasks dynamically outside loops. This gives CFlex the power to parallelize accesses to lists, trees and other pointer-based structures as well as recursive algorithms. This last ability gives CFlex an advantage over OpenMP, which can only express the parallelization of iterative constructs.

3.1.2 Critical Sections

The `critical (`*`name`*`)` directive declares the (possibly compound) statement following it as a critical section. All the critical sections with the same name are mutually exclusive. This is achieved through the use of the same FXCC lock for all of them.

This clause implies a synchronization among processors to work on some shared data that may be modified. Thus it requires invalidating the caches of the processors and before entering the critical section so that they are forced to read the latest version of the shared data. Also, the cache must be flushed when the processor exits the section, so that the next processor to enter it will be able to access the (possibly) new versions of those data. Performance may be improved by using the optional clause `flush` to specify which are the pieces of data that require the invalidation on the entry and the flush on the exit of the section.

3.1.3 Miscellanea

Two other execution modifiers have been implemented for research purposes:

- `fast` runs the statement following it in fast mode in our simulator.
- `time(`*`expr`*`)` measures the number of cycles required to execute the following statatement in our simulated system. The expression must yield an integer value that will be printed next to the number of measured cycles. Its purpose is to identify the piece of code that has been timed.

3.2 Data Modifiers

The following data modifiers have been implemented:

- `alignable` precedes the declaration of a C `struct` or `union`. It adds a padding field so that the size of the data structure will be a power of two, and thus alignable.
- `page_aligned` works very much like `alignable`, but the padding field size is calculated so that the size of the resulting data structure will be a multiple of the page size.
- `align (`*`align_spec`*$_1$`, `*`align_spec`*$_2$`, ...)` tries to align the dimensions of vectors and arrays. The syntax of each alignment specification is given by *array_name*`([])*`, where the number of square brackets pairs specifies which is the dimension to align. The compiler will turn the size of each element of that dimension into a multiple of the page size by padding the smaller dimensions of the array.

```
#pragma FlexRAM phost sync
  for(p = head; p != NULL; p = p->next)
#pragma FlexRAM parray async on_home(*(p->data)) firstprivate(p)
    process(p->data);
```

Figure 3: Parallelized linked list processing using the `sync` and `async` clauses

```
#pragma FlexRAM parray pfor on_home(*(p->data)) firstprivate(p)
  for(p = head; p != NULL; p = p->next)
    process(p->data);
```

Figure 4: Parallelized linked list processing using the `pfor` clause

3.3 Executable Directives

- `flush(`exp_1`, `$exp2$`, ...)` orders flushing a series of data items from the cache of the processor. An optional clause `invalidate` tells besides to invalidate them after the flush.
- `barrier(`*`expr`*`)` implements a barrier of *expr* PArrays.
- `debug(`*`debug_expr`*`)` causes the simulator to print *debug_expr*, which must yield an integer value, in debugging mode. An optional clause `mark(`*`expr`*`)` prints also the integer value *expr* as mark to distinguish among several possible data items being debugged.

4 Examples of the use of CFlex

Our basic description of CFlex provides enough context to illustrate its flexibility with some examples. Consider the code in Fig. 3. The original task, say T_1, generates a task, T_2, corresponding to the loop. Task T_1 will wait until task T_2 completes. Task T_2 spawns one task for each iteration of the loop that traverses the linked list. Since these tasks are asynchronous, T_2 does not wait for them to complete and continues spawning without pause until in has completed all loop iterations. The task corresponding to each iteration of the loop will be run in the PArray on whose bank `p->data` is located and it will receive a privatized copy of the value of pointer `p` for that iteration. The `for` loop (task T_2) is marked as a synchronous task in the PHost. As a result, the compiler will insert code to wait for the tasks generated in the PArray(s) to finish before T_1 continues with the code following the loop. It is interesting to note how this scheme allows the specification of overlapping tasks in the PHost and the PArrays in a natural way just by adding more code to be run by the PHost inside the synchronous task.

As can be seen in this example, loops can be parallelized by enclosing the whole loop inside a synchronous task that will provide the required synchronization point, and by specifying that each iteration of the body is an asynchronous task. Loops are the most common source of parallelism, so CFlex has been extended with a `pfor` clause that tells the compiler to break the loop following it into a series of tasks and wait for those tasks to finish before continuing. The previous loop rewritten using a `pfor` clause is shown in Fig. 4.

A more elaborate parallelization scheme is required when there are portions of code that must be run sometimes in the PHost, and sometimes in the PArrays. Fig. 5 shows an example of this kind extracted from the `treeadd` code, part of the Olden benchmark suite [14]. This routine adds the values stored in the nodes of a tree. During the construction of the tree, the upper levels were

```
int TreeAdd (register tree_t *t) {
  if (t == NULL) return 0;
  else {
    int leftval, rightval, value;
    tree_t *tleft, *tright;

#pragma FlexRAM phost sync
    {
      tleft = t->left;

#pragma FlexRAM parray async on_home(*tleft) if (lclHeapProcId(tleft))
#pragma FlexRAM phost  async else
      leftval = TreeAdd(tleft);

      tright = t->right;

#pragma FlexRAM parray async on_home(*tright) if (lclHeapProcId(tright))
      rightval = TreeAdd(tright);
    }
    value = t->val;
    return leftval + rightval + value;
  }
}
```

Figure 5: Parallelized tree processing

allocated by the PHost while the subtrees below a given level were allocated by the PArrays. Our runtime system allows all the PArrays to allocate and deallocate heap memory in parallel.

The function `lclHeapProcId`, which is part of this runtime system, returns zero if a node was allocated by the PHost and a positive number if it was allocated by a PArray. The routine proceeds by spawning one PHost task for each of the left children in the upper levels of the tree (those allocated by the PHost and therefore with an `lclHeapProcId` value of zero). Once a PArray task is spawned, no new tasks will be created since PArrays are incapable of creating new tasks. Therefore, once a PArray task is spawned, the whole subtree will be processed sequentially.

Because PArrays cannot create tasks, the compiler generates two versions of this kind of routines. The PHost version includes all the tests and the task data generation, spawn and synchronization calls, while the PArray version only contains the original code of the routine.

Notice also that in both examples it was not necessary to add or modify any line of the sequential code. All that was needed to parallelize the code was to insert directives. This is typical of the codes we have parallelized for FlexRAM.

5 Software support

The management of our PIM system requires a series of extensions to the Operating System (OS) as well as a run-time system. We also propose the use of libraries that allow to exploit the memory processors while hiding their programming. The following sections ellaborate on these subjects.

5.1 Operating System

One of our extensions to a standard Operating System that when a page is referenced for the first time and that reference comes from a PArray, our Operating System places the page in the memory bank of that PArray. Also, our OS must ensure that the PArrays do not fall in an incorrect state

when a page replacement is required. Currently this is achieved by precluding the replacement of pages containing information used by the PArrays. In the future, we plan to enable the replacement of these pages by implementing operations to flush all the references to them from the caches and TLBs of the PArrays. The implementation could be based on a table containing the list of PArrays that point to each physical page in their TLB. When a PArray suffered a TLB miss that led to the replacement of the entry associated to page P, it would remove itself from the list associated to P, and it would flush the data of this page from its cache. Then it would add itself to the list of PArrays that keep the mapping for the page that generated the TLB miss. These two steps could be restricted to only those pages that are not already in the PArray bank, as we expect the PArrays to generate most of their references to the local bank. Our OS would use these tables to know which PArrays need to be instructed about the replacement of a given page in order to flush the data of the page and invalidate the corresponding mapping. Notice that this table would be also very useful to implement the page migration mechanism because only the specified processors would have to be informed about the migration. The FXCC locks introduced in Sect. 2.1 would provide for the required mutual exclusion during the operation of the OSs of the different processors.

Our simulated Os exports the variables `NumPagesPerBank`, `NumBanksPerChip`, `NumBanks`, `NumFlexRAMChips` and `NumPArrays`, whose meanings should be obvious. Besides, the following functions have been implemented in our OS and maybe useful to program the system:

- `int getProcessorId(void)` returns a unique identifier for each processor, being 0 the PHost, 1 the first PArray, etc.
- `flushCache(int invalidate)` invalidates all the caches of the calling processor. If `invalidate`$\neq 0$ the caches are also invalidated.
- `void flushAddrRangeCache(void *addr, unsigned long int size, int invalidate)` flushes from the caches of the calling processor the memory region the begins in address `addr` and extends for `size` bytes. That memory region can also be invalidated if `invalidate`$\neq 0$.
- `void bootFlexRAM(void)` boots the FlexRAM system including the PArrays Operating Systems, the FlexRAM chips periodical polling, etc.
- `void exitFlexRAM(int status)` terminates the PArrays Operating Systems, the periodical activities, etc. and returns `status` as exit code.
- `void OSTmpSuspend(long cycles)` suspends the execution of the calling thread for (at least) `cycles` cycles.

5.2 Run-time System

A runtime system implemented as a dynamically linked library would take care of those software procedures that do not require OS support. Those functions include:

- an interface to the FXCC locks and constructions built upon them, like barriers.
- heap memory management including parallel allocation and deallocation of space.
- the polling of the FlexRAM chips to detect requests from the PArrays (see Sect. 2.1). The OS might have to be invoked to serve some requests such as page faults.

These are the interfaces to the functions that could be useful for the programmer:

- `void FxRAMPHSpawnTask(void (*func)(void*), void *arg)` spawns a new thread on the PHost that will execute function `func` with argument `arg`. This `arg` is a pointer to a memory region whose first component must be a `FxRAMTaskDescriptor` structure:

```
typedef struct FxRAMTaskDescriptor {
  unsigned short int code_task;
  short int sync_flag;
  unsigned short int parray_num;
  struct FxRAMTaskDescriptor *next;
} FxRAMTaskDescriptor;
```

- `void FxRAMPASpawnTask(void (*func)(void*), void *arg, unsigned short int fxchip, unsigned short int parray)` spawns the task `func` with argument `arg` on the PArray `parray` in the FlexRAM chip `fxchip`. The value `USHRT_MAX` means "any" for these two arguments. This `arg` is a pointer to a memory region whose first component must be a `FxRAMTaskDescriptor` structure.

- `void FxRAMLinkTask(FxRAMTaskCtrl *ctrl, FxRAMTaskDescriptor *descr, unsigned short int code_task)` links the task defined by `descr` to the task control structure `ctrl`, defined as:

```
typedef struct {
  unsigned int num_tasks;
  FxRAMTaskDescriptor *head, *tail;
} FxRAMTaskCtrl;
```

  The argument `code_task` is the value that will be assigned to the `code_task` field of the `descr` structure.

- `unsigned short int FxRAMWaitTasks(FxRAMTaskCtrl *ctrl, void **arg)` waits for the first task registered in the task control structure `ctrl`. This function returns in `arg` a pointer to the memory region that was passed to the task for possible recovery of outputs/deallocation of that memory. The return value of the function is the `code_task` field that was associated to this task.

- `void FxRAMChainedSpawn(FxRAMTaskDescriptor *p, FxRAMTaskDescriptor *head, void (*func)(void*))` spawns on PArrays the task associated to `p` and all of its following tasks (in the list linked by the pointer `next` of this structure) till `head`. The function to execute will be `func` in all of them.

- `unsigned short int getHomePArray(void *data)` returns which is the PArray on whose bank the data `data` is located. It returns `USHRT_MAX` if that data is not located in the bank of any PArray.

- `int preMap(void *data, unsigned long int size)` accesses all the pages in the memory region that starts in address `data` and that extends for `size` bytes. This function ensures that all the memory region is mapped in memory before the PArrays begin to work on it.

- `void *localMalloc(unsigned int size)` allocates a region of `size` bytes in the piece of the heap that is locally managed by the Operating System of the calling processor.

Function description	Abstract expression	Syntax
Apply func `f` with arg `a`	exec `f(v(i),a)`, `0 <i<N`	`Vector_apply(v,f,a)`
Search element that fulfills condition `f` with arg `a`	ret any `v(i)` such that `f(v(i),a)`$\neq$ 0, `0<i<N`	`Vector_search(v,f,a)`
Generate vector with the result of applying func `f` with arg `a`	`v2(i)` = `f(v(i),a)` `0<i<N`	`v2=Vector_map(v,f,a)`
Reduce vector applying func `f`, whose neutrum is `a`	ret `f(...f(a,v(0))...,v(N-1))` where `f(a,x)=x`	`Vector_reduce(v,f,a)`
Process together two vectors and an arg `a`, generating a new vector	`v3(i)` = `f(v(i),v2(i), a)` `0<i<N`	`v3=Vector_map2(v,v2,f,a)`

Table 1: Some Vector container IMOs

- `void localFree(void *ptr)` deallocates the memory region that begins in the address `ptr`. This region must be part of the piece of the heap that is locally managed by the Operating System of the calling processor.
- `unsigned short int lclHeapProcId(void *addr)` identifies which is the processor on whose locally managed heap the memory position `addr` is located.
- `void FxRAMAcquireLock(void *lock)` tries to acquire the FXCC lock whose token is given by `lock`.
- `void FxRAMReleaseLock(void *)` releases the FXCC lock whose token is given by `lock`.
- `void FxRAMBarrier(volatile int *a, volatile int *b, int n)` implements a barrier among `n` processors using the FXCC locks `a` and `b`.
- `unsigned short int getLogicalProcessorId(void)` returns the logical processor id of the task. If the calling task has resulted from the parallelization of a loop, this is a value between 0 and the number of tasks in which the loop has been parallelized. Ohterwise it will return 0.

5.3 Intelligent Memory Operations

Although the CFlex pragmas are all we need to program our system, programmability can be highly improved by providing libraries of subroutines that make use of the PArrays but that hide completely their existence. We call these subroutines *Intelligent Memory Operations (IMOs)*. The IMOs may work on data structures defined by the user performing typical operations on them. Examples of this kind of IMOs would be finding the minimum value in a vector and adding two matrices. Other IMOs can be envisioned like STL classes [15] that make use of the PIMs throughout all the phases of the life of a given data structure. For example, IMO STL classes may define and operate on containers such as lists, hashes, sets, etc. making use of the PIMs to perform in parallel allocations and deallocations of elements, searches, and other computations with the elements stored in these containers. Some IMO functions for a vector container are proposed in Table 1.

6 Evaluation

The performance evaluation of our hardware/software system, including our compiler, is based on a cycle-by-cycle detailed execution driven simulator designed by our research group. Our baseline is a standard workstation with a 1.6 GHz five-issue high-performance processor similar to the Power4. We measure the speedup achieved when its plain memory modules are replaced with FlexRAM chips

PHost processor	PHost caches	Bus & Memory
Freq: 1.6 GHz	L1 size: 32 KB	Bus: split transaction
Issue width: 5	L1 OC,RT: 1,3	Bus width: 16 B
Dyn issue: yes	L1 assoc: 2	Bus freq: 400 MHz
I-window size: 64	L1 line: 128 B	Mem: 1-channel Rambus
Ld/St units: 2,1	L1 HUM: 16	Mem RT: 180 (112.5 ns.)
Int,FP units: 5,4	L2 size: 1 MB	
Pending Ld,St: 32,32	L2 OC,RT: 4,12	
BR penalty: 12	L2 assoc: 8	
TLB entries: 128	L2 line: 128 B	
	L2 HUM: 8	
PArray processor	**PArray Cache**	**FlexRAM torus & bus**
Freq: 1 .6GHz	L1 size: 8 KB	Torus width: 8B
Issue width: 2	L1 OC,RT:1,2	Torus freq: 1.6 GHz
Dyn issue: no	L1 assoc: 2	Bus: split transaction
Ld/St units: 1,1	L1 line: 32 B	Bus width: 16 B
Int,FP units: 1,1	L1 HUM: 0	Bus freq: 400 MHz
Pending Ld,St: 2,2		
BR penalty: 6		
TLB entries: 32		

Table 2: Parameters of the architecture simulated. OC, RT and HUM stand for occupancy, round trip from the processor and hit under miss, respectively.

Applic.	Problem size		Original lines	Directives	Additional lines
	Nodes	Memory			
TSP	512 K	22 MB	485	12	5
TreeAdd	2 M	40 MB	71	8	4
Equake	7 K	3.6 MB	1227	14	6
Matrix	1000x1000	24 MB	81	1	2
Distance	30 K	1.2 MB	108	17	7
Path	400 K	12.8 MB	165	17	9
Average	991K	17.26 MB	356	11.5	5.5

Table 3: Application characteristics

programmed using CFlex and our runtime system. The main related architectural parameters are listed in Table 2, where all the times are measured in PHost cycles. Notice that the main processor of the system is extremely powerful, has a large L2 cache and is able to sustain many simultaneous accesses in parallel.

The applications and problem sizes used in our evaluation are described in Table 3. TSP and TreeAdd are two well-known Olden benchmarks [14]. Equake is an SPEC OMP2001 run with the test input set. Matrix is a dense matrix product with blocking in the three dimensions that uses 1000x1000 double precision matrices. Finally, Distance and Path are two applications extracted from [16] that make use of an IMO implementing a singly-linked list. The last three columns of Table 3 attempt to measure the effort required to parallelize the applications by showing their original number of lines of code as well as the number of CFlex directives and additional lines of code required to generate the parallel version. Little effort has been made to optimize the parallel version, other than using a strategy that distributes the tasks evenly among the PArrays. Rather, we have stressed the simplicity of the parallelization scheme by making as few modifications as possible, as the figures in the table show.

The parallelization of the two Olden benchmarks is similar: they both build a tree and then

perform some kind of processing on it. Both stages of these programs have been parallelized by choosing a level in the tree under which each one of the subtrees is processed by a single PArray. This scheme provides a good load balance between the PArrays. An example corresponding to the tree reduction in TreeAdd can be seen in Fig. 5. The upper levels of the tree are thus processed by fewer PArrays as the number of subtrees is reduced in TSP. The PHost is in charge of processing all the nodes of the uppers levels in TreeAdd because of the very short computation necessary per node.

As for Equake, it has been parallelized by replacing the OpenMP pragmas provided in the SPEC OMP suite with CFlex pragmas. It deserves mention that it allocates 290 additional KB of memory per each new PArray thread. Consequently, its memory usage grows with the degree of parallelization.

The matrix product has been parallelized simply by instructing the system to calculate the result for each block of rows of the destination matrix in a different PArray. This way, only one pragma is required to split this parallel loop.

In the case of Distance and Path we have not started from a sequential version. Instead, we have written the programs making use of the IMOs. Thus, for these applications we show the total number of CFlex directives inside the IMO library (Directives column) and the static number of calls to IMO functions in these applications (Additional lines). Also, the original code size reflects the number of lines of the applications without including the IMO library, which is 714 lines long, as it is provided by the system. The IMO functions are designed to be near-optimal both for the sequential and the parallel execution. Still, there are particular cases in which the sequential execution performance can be hindered by the structure of the code oriented to the parallel execution. In those cases, we have written versions of these functions that are optimized for the sequential execution, and we use them as our baseline when the code is run on the PHost.

6.1 Speedups

The speedups achieved for these applications using one or two FlexRAM chips over the plain system as well as the corresponding geometric means are shown in Fig. 6. We see that despite the simple parallelization schemes, very good speedups have been achieved for most of the codes. This is particularly encouraging when we consider the simplicity of our PArrays compared to the main processor of the system.

It is interesting to see how some of the applications benefit from the usage of several chips while others do not. The main reason for this behavior difference lies in the degree of locality in the accesses to memory by the PArrays. Specifically, the use of several FlexRAM chips usually implies communication through the FlexRAM bus, which turns the bus into a source of contention when the locality of the PArrays is poor, and they need to access data in banks located in other chips. For example, in TSP each PArray can initially process locally a subtree of the lower part of the tree. However, as the processing of the tree goes up, there are fewer subtrees, which means both less parallelism and that the PArrays have to access data in more and more banks. The performance penalty is higher when some of those banks are located in another chip. This is the reason for the small improvement as we go to two FlexRAM chips in TSP.

TreeAdd does not suffer from this problem because its parallelization allows each PArray to process exclusively data located in its bank, and the reduction of the data provided by the PArrays can be quickly performed by the PHost.

The bus bottleneck problem is the worst for very regular parallel codes in which many processors request almost simultaneously certain pieces of data, thus turning the corresponding banks into sources of contention too. This is what happens in the dense matrix product.

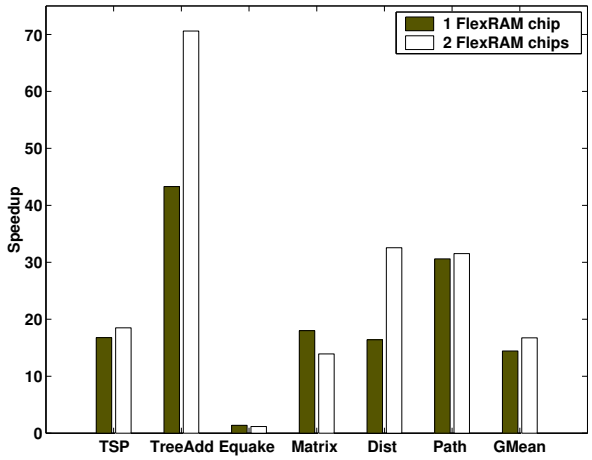

Figure 6: Speedups obtained using FlexRAM chips.

There are situations in which although the parallelization scheme and the distribution of the data are both good, there is just not enough data or processing to justify splitting the work among more processors. The advantages obtained by the use of more PArrays vanish due to the growth of the overheads, in Equake and Path. There is an additional reason for the relatively small speedups of Equake: the maximum theoretical speedup it can achieve for the test input set is 2.13, as only 53% of its time is spent in the parallelizable loops. Our parallelization reduces the execution time of the parallel loops to about half, achieving a speedup of 1.38 with one FlexRAM chip and 1.18 with two chips.

7 Conclusions

This paper addressed the issue of providing ease of use and efficient programming support for FlexRAM [7], an intelligent memory architecture. We know of no other works focused on solving the programmability issues related to this kind of systems. This task required solving several challenges, as FlexRAM imposes a large number of restrictions. On the one hand, its memory processors lack a large number of interprocessor synchronization and communication mechanisms and there is very little hardware support for cache coherence. On the other hand, the memory processors depend on the main processor(s) of the system to perform several tasks. These problems have been overcome by combining CFlex, a novel family of pragmas inspired on OpenMP, with a run-time system. Our directives are adapted to the nature of our PIM chips and they allow simple and effective parallelization of a much larger class of problems than the current standard OpenMP does. The complexity of the system can be further hidden by providing libraries of functions that operate on data structures making use of the PIMs but without requiring the programmer to have any knowledge of them. Such functions, called Intelligent Memory Operations (IMOs), optimize the usage of the system while reducing the programming effort. Our experiments show that a workstation with PIM chips can easily achieve speedups of over one order of magnitude with few changes to the source code of the applications.

References

[1] IBM Microelectronics: Blue Logic SA-27E ASIC. http://www.chips.ibm.com/news/1999/sa27e

(1999)

[2] Iyer, S., Kalter, H.: Embedded DRAM technology: opportunities and challenges. IEEE Spectrum (1999)

[3] Patterson, D., et al.: A Case for Intelligent DRAM. IEEE Micro (1997) 33–44

[4] Waingold, E., et al.: Baring It All to Software: Raw Machines. IEEE Computer (1997) 86–93

[5] Rixner, S., et al.: A Bandwidth-Efficient Architecture for Media Processing. In: International Symposium on Microarchitecture. (1998)

[6] Kaxiras, S., Sugumar, R., Schwarzmeier, J.: Distributed Vector Architecture: Beyond a Single Vector-IRAM. In: First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. (1997)

[7] Kang, Y., Huang, W., Yoo, S., Keen, D., Ge, Z., Lam, V., Pattnaik, P., Torrellas, J.: FlexRAM: Toward an Advanced Intelligent Memory System. In: International Conference on Computer Design. (1999) 192–201

[8] Oskin, M., Chong, F., Sherwood, T.: Active Pages: A Computation Model for Intelligent Memory. In: International Symposium on Computer Architecture. (1998) 192–203

[9] Hall, M., et al.: Mapping Irregular Aplications to DIVA, a PIM-based Data-Intensive Architecture. In: Supercomputing. (1999)

[10] OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface Version 2.0. (2002)

[11] Crisp, R.: Direct Rambus Technology: the New Main Memory Standard. In: IEEE Micro. (1997) 18–28

[12] Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.A.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. SIGPLAN Notices **29** (1994) 31–37

[13] Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C.A., Offner, C.D.: Extending OpenMP for NUMA Machines. In ACM, ed.: SC2000: High Performance Networking and Computing, ACM Press and IEEE Computer Society Press (2000) 68–69

[14] Rogers, A., Carlisle, M.C., Reppy, J.H., Hendren, L.J.: Supporting Dynamic Data Structures on Distributed-Memory Machines. ACM Transactions on Programming Languages and Systems **17** (1995) 233–263

[15] Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project (1994)

[16] Foster, C.: Content Addressable Parallel Processors. Van Nostrand Reinhold Co, New York (1976)