Programming the FlexRAM Parallel Intelligent Memory System[*]

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

Dept. de Electrónica e Sistemas, Universidade da Coruña, Spain
basilio@udc.es

[†]Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA
{renau,padua,torrellas}@cs.uiuc.edu

[‡]LIP, Ecole Normale Supérieure de Lyon, France
Paul.Feautrier@ens-lyon.fr

ABSTRACT

Intelligent memory architectures enhance the memory chips of a computer with many simple processors. The result is a highly-parallel, heterogeneous machine that is able to exploit computation in memory. Examples of such architectures are FlexRAM, DIVA, and Active Pages.

In this paper, we address how to effectively hand-program such an architecture. We propose a family of compiler directives inspired by OpenMP called CFlex. Such directives enable the memory processors to cooperately execute the program with the main processor. In addition, we propose libraries of highly-optimized functions called Intelligent Memory Operations (IMOs). These functions program the processors in memory through CFlex, but make them completely transparent to the programmer. Simulation results show that, with CFlex, a server with intelligent memory often delivers a performance that is 10 times higher (or more) than a plain server.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

[*]This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; by the Ministry of Science and Technology of Spain under contract TIC2001-3694-C02-02; and by gifts from IBM and Intel.

Keywords

Intelligent memory architecture, compiler directives, programming heterogeneous computers, parallel languages.

1. INTRODUCTION

Integrating processors and main memory in the same chip is a promising approach to address the processor-memory communication bottleneck. In such chips, processors enjoy short-latency and high-bandwidth communication with memory. One way to use these chips is as intelligent memories that replace all or some of the standard memory chips in a server or workstation. This is the approach followed by the FlexRAM [11], DIVA [6], and Active Pages [14] intelligent memory systems.

This use of processor-memory chips as intelligent memory is very appealing because it requires relatively few changes to general-purpose computers, and it supports the execution of applications without modifications. Indeed, applications can be gradually modified or compilers gradually improved to take advantage of processing-in-memory capabilities.

Unfortunately, the challenging problem of effectively programming such machines has received only limited attention. Specifically, there are some proposals where the programmer identifies and isolates the code sections to run on the memory processors [2, 6, 11, 14]. However, these proposals are largely concerned with dividing sections of code across a set of identical memory processors and lack portability. The resulting approach is often not much different from running code on a conventional parallel processor.

An alternative approach has been to use a compiler that automatically partitions the code into loops and other sections and then schedules each section on either the main processor or the one in memory [16]. This approach has only been tried for a system with a single processor in memory; it has not been applied to a general heterogeneous system with several main processors and many memory processors.

In this paper, we present a language and necessary operating/run-time system support to enable the efficient programming of such a general heterogeneous architecture. Our goal is to provide language support to develop highly-tuned applications that are relatively easy to understand and modify. To this end, we devise CFlex, a set of directives resembling those of OpenMP [13] that control execution on an intelligent memory system. CFlex exposes the intelligent memory architecture to the programmer, therefore unlocking the performance potential of the system. Moreover, the use of directives results in portable programs, which can be compiled for execution on plain systems by simply ignoring the directives.

Since applications should also be able to profit from intelligent memory without the programmer having to be concerned with the architecture organization, we briefly study libraries of Intelligent Memory Operations (IMOs). They are written using CFlex and hide from the programmer the organization of the intelligent memory.

Our discussion of CFlex, its implementation, use, and evaluation are all made in the context of the FlexRAM intelligent memory architecture [11]. In this environment, our simulation results show that, with CFlex, a server with intelligent memory often delivers a performance that is 10 times higher (or more) than a plain server.

The rest of this paper is organized as follows: Section 2 outlines the FlexRAM architecture; Section 3 describes the relevant operating and run-time system support; Section 4 describes CFlex and provides illustrative examples; Section 5 discusses IMO libraries; Section 6 shows how CFlex can be applied to other intelligent memory architectures; Section 7 presents the environment that we use to evaluate CFlex on FlexRAM; Section 8 presents the evaluation; Section 9 discusses related work; and Section 10 concludes.

2. FLEXRAM ARCHITECTURE

A FlexRAM system is an off-the-shelf workstation or server where some of the DRAM chips in the main memory are replaced by FlexRAM processor-memory chips [11]. Each FlexRAM chip contains DRAM memory plus many simple, general-purpose processing elements called *PArrays*. To the main processor of the system, which we call *PHost*, the resulting memory system appears as a versatile accelerator that can off-load memory-intensive or highly-parallel computation. While the machine can have multiple PHosts, in this paper we consider a single-PHost system. Each PArray can be programmed independently and, therefore, PArrays can execute SPMD or MIMD code. The PHost and all PArrays share a single address space. Finally, if the *PHost* runs legacy applications, the memory system appears as plain memory. A FlexRAM system is shown in Figure 1.

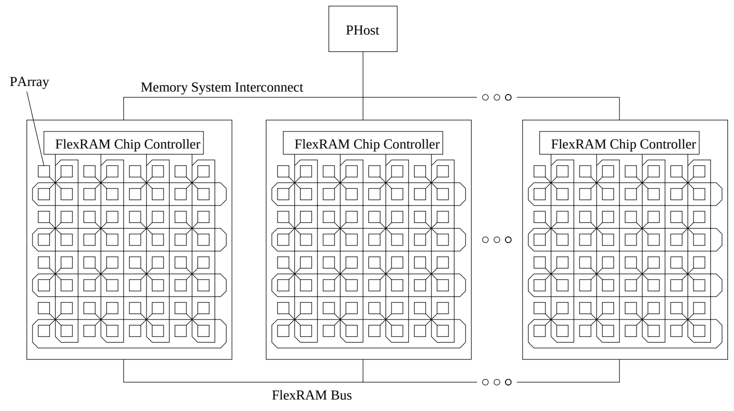

Figure 1: A FlexRAM intelligent memory system.

The architectural parameters of a FlexRAM chip have been upgraded from [11]; the new parameters are described in [19]. In this upgraded design, each FlexRAM chip has 64 PArrays, a FlexRAM chip controller (*FXCC*) that interfaces the PArrays to the PHost, and 64 Mbytes of DRAM organized in 64 banks. Each PArray has an 8-Kbyte write-back data cache and a 2-Kbyte instruction cache (Figure 2). A FlexRAM chip has a 2-D torus that connects all the on-chip PArrays. Moreover, all the FlexRAM chips are connected to a communication bus called the FlexRAM bus. With these links, a PArray can access any of the memory banks in its chip and in other FlexRAM chips.

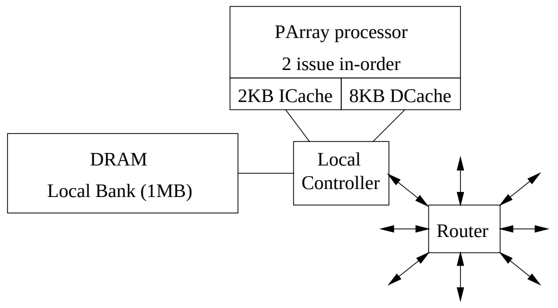

Figure 2: Structure of a PArray and a memory bank inside a FlexRAM chip.

PArrays use virtual addresses in the virtual address space of the application run by the PHost. Each PArray has a small TLB that contains some entries from the PHost's page table. PArrays serve their own TLB misses. However, they invoke the PHost operating system for page faults and migrations.

2.1 Interprocessor Communication

Since the memory system interconnect of a FlexRAM machine is off-the-shelf, FlexRAM chips cannot be masters of it. As a result, any communication between PArrays and PHost has to be done in one of two ways: through memory or via memory-mapped registers in the FXCCs. Memory communication involves writing and reading a memory location, and is typically used to pass the input arguments and outputs of the tasks executed on the PArrays. FXCC register communication involves writing and reading special registers in the FXCCs, and is used for commands and service requests. We consider FXCC register communication next.

The PHost communicates with the PArrays to spawn tasks on them, answer service requests, and order maintenance operations such as flushing cached pages or invalidating TLB entries. In all cases, the PHost issues a command to an FXCC register and passes at most two words, namely the address of the routine to execute and a pointer to the input argument buffer. The FXCC stores this information and passes it to the corresponding PArray(s).

A PArray communicates with the PHost to request a service, such as the handling of a page fault. In this case, the PArray writes a register in its FXCC. The FXCC cannot deliver this request to the PHost because FlexRAM chips cannot be masters of the memory system interconnect. Consequently, the PHost periodically polls the FXCCs to check for requests.

2.2 Synchronization

Each FXCC manages a set of locks that can be acquired and released by the PHost, and by the on- and off-chip PArrays. Upon a request for a free lock, the home FXCC grants ownership to the requester. If the lock is currently taken, the FXCC is able to queue up a certain number of requests, and reply to each of them when it can grant ownership. If the number of requesters exceeds a certain threshold, the FXCC replies back with a "busy" message.

2.3 Data Coherence

A FlexRAM system lacks hardware support for cache coherence between the PHost and PArrays. Data coherence is maintained by explicit (total or partial) cache writeback and invalidation commands [16]. Specifically, before the PHost initiates a PArray task, it writes back the dirty lines in its caches that contain data that may be read by the PArray task. This ensures that the PArray sees the correct version of the data. Moreover, before the PHost executes code that may use results from a task executed by a PArray, the PHost invalidates from its caches the lines with data that may have been updated by the PArray. This ensures that the PHost does not use stale data. As for a PArray, when it completes a task, it writes back its dirty cache lines and invalidates its small cache. The PArray caches also include a dirty bit per word, so that only the modified words actually update memory. This is done to tolerate false sharing between PArrays.

3. OPERATING & RUN-TIME SYSTEM

To use the FlexRAM system, we need several extensions to the Operating System (OS) of the PHost and a small per-PArray kernel. In general, the PHost OS is in charge of all the I/O operations, PHost CPU scheduling, and virtual memory management. The PArray kernel manages the PArray's TLB, and the spawn and termination of local tasks. In addition, we have developed a library-based user-level run-time system for both the PHost and PArrays to improve the programmability of the machine. In the following, we describe the management of virtual memory and the user-level run-time system.

3.1 Managing Virtual Memory

PHost and PArrays share a common view of the address space of the tasks that cooperate in the execution of an application. The PArray kernel reads the page table of the process and updates its local TLB. However, only the PHost OS can update the page table, perform page swapping, or migrate pages.

To maximize access locality, the PHost OS tries to map the virtual pages that a PArray references, to physical pages located in the PArray's local memory bank. Currently, we support this approach with a "first-touch" mapping policy, where the first PArray to reference a given page becomes its owner and allocates it locally.

The PHost OS cooperates with the PArray kernels to keep the PArray TLBs coherent when the page table changes. Specifically, there is a shared software structure that contains, for each page mapping, the list of PArrays that cache it in their TLBs. This structure is updated by the PArray kernels as they update their TLBs. When the PHost OS moves a page to disk, it informs all the PArrays that cache the mapping of that page in their TLBs. Then, the corresponding kernels invalidate that TLB entry and the relevant cache lines. The structure described is also used in other situations when the PHost OS modifies the page table, such as when it migrates pages.

3.2 User-Level Run-Time System

We developed a library-based user-level run-time system for both the PHost and PArrays to perform several functions. These functions include task management, synchronization, heap memory management, and polling for requests. We consider these functions in turn.

Tasks are spawned on the PArrays by the user-level run-time system. Each task has an associated software buffer that contains the task input data that is not shared and, sometimes, the results of the task. The former includes private copies of global variables that the task needs; the latter is not necessary if all the results are stored in shared memory structures. To start a task, the user-level run-time system fills the inputs in this buffer and then sends a spawn message to the FXCC of the FlexRAM chip where the task is to run. The PHost can then use the run-time system to spin on a location in the buffer that the PArray task will set when it finishes.

The run-time system also includes synchronization routines that use the FXCC locks, often building higher-level constructs such as barriers.

The run-time system also performs heap memory management, including parallel allocation and deallocation of heap space by the PArray tasks. Standard `malloc` and `free` functions are used, both in the PHost and PArrays. Those in the PArrays operate on local pages of the heap that the run-time system in the PHost has allocated, assigned to that particular PArray, and requested that the OS map them in that particular PArray's local memory bank. When the PArray task runs out of local heap, it asks for more to the PHost run-time system.

Finally, another function of the run-time system in the PHost is to poll the FXCCs to detect requests from the PArrays.

Note that the run-time system exports its functions and variables not only to the compiler but also to the programmer. The complete description of its interface is found in [5].

4. CFLEX

To exploit the functionality of FlexRAM while keeping the portability of the programs, we program the FlexRAM system using a family of compiler directives. Our computational model requires these directives to be able to express the task partitioning between the PHost and the PArrays, and the synchronization between the generated tasks. Other desirable properties are scalability as the number of processors available in memory increases, and hiding as much system details as possible from the programmer. It is also important that the directives be powerful and flexible, so that they allow the generation of parallel programs that are easy to read and modify for a wide class of problems.

The latter objective is essential for our environment because of the nature of the FlexRAM system architecture. Specifically, the PArrays are much simpler than the PHost, have a lower clock rate, and lack floating-point units. This means that they are not particularly well suited to speed up typical parallel applications based on floating-point operations and loops that use regular, cache-friendly data structures. Instead, the PArrays's most valuable property is their low memory access time, which allows them to accelerate irregular, memory-intensive applications. This is an interesting challenge for the design of our family of directives. Indeed, currently available parallelization directives such as OpenMP [13] or HPF [8], are especially oriented to loop-parallel codes, and not well suited for irregular applications like those using pointers or indirect accesses.

CFlex is a family of directives that addresses these issues. The structure of CFlex directives is similar to that of OpenMP directives, but there are also differences. We implemented CFlex as annotations to the C language, and use C in the discussion below. However, CFlex can be easily used with other languages like FORTRAN.

The general structure of a CFlex directive is

#pragma FlexRAM *`directive-type [clauses]`*

CFlex directives may be classified in three groups:

- *Execution modifiers* indicate how a given segment of 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* request that data structures satisfy certain conditions. An example are directives to pad one of the dimensions of an array to make its size a multiple of the page size.

- *Executable directives* appear like instructions in the program. Examples of this class include barriers or prefetch operations.

In the following, we briefly discuss the three kinds of directives and then show some examples of their use. A complete description of CFlex can be found in [5].

4.1 Execution Modifiers

These directives partition the computation into PHost and PArray tasks, and synchronize tasks. For example, one important execution modifier specifies the kind of processor where the next statement should be executed. It does so by setting *directive-type* to either `phost` or `parray`. Since PArrays cannot spawn new tasks, these directives may only appear in the code executed by the PHost, and they will spawn either a new PHost task or a PArray task. A series of optional *clauses* enrich the semantics of the directive:

- `on_home(`*x*`)` specifies that the task should be executed on the PArray whose local memory bank contains the *x* data structure. If the *directive-type* is `phost`, then this clause only has effect when the target computer is a NUMA machine. In this case, it specifies that the new task should be run on the PHost whose local memory contains *x*.

- `sync`/`async` specifies whether the task spawning the new task must stop until the new task finishes (`sync`) or it can continue (`async`). The default is `sync`. A task does not finish until all the tasks that it has spawned have finished. CFlex only allows the creation of asynchronous tasks inside the syntactical scope of a synchronous task. This provides a clear point of synchronization where all asynchronous tasks are known to have finished. This point is the end of the synchronous task inside which the asynchronous ones have been spawned. This approach is illustrated in Section 4.4.

- `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. Note that execution modifiers cannot be nested for the very same piece of code. Therefore, there is no need to implement symmetric conditionals.

- `shared`, `private`, `lastprivate`, `firstprivate`, `reduction` have the same semantics as the directives of the same name in OpenMP. In contrast to OpenMP, CFlex allows them to apply to only one part of a structure, such as one field of a struct or one element of a vector.

- `flush` specifies variables such that, if their corresponding lines are dirty in the PHost cache, the lines are written back to memory. This is done so that the PArray(s) executing the task(s) have access to the latest version of the data they require in memory. If this clause is not present, the compiler writes back all the dirty lines in the PHost cache. The programmer can use this clause to improve the performance by restricting the writeback to a certain set of variables.

Note that the ability to spawn new PHost tasks allows CFlex to express parallelism in systems without processing in memory. In this case, the `on_home` clause is useful to control locality of execution when the PHosts belong to a NUMA machine.

A `migrate` clause could be added in the future to designate shared data structures whose pages should migrate to the first PArray or PHost that touches them after the task(s) created by this directive begin their execution. However, page migration can be very costly.

A second execution modifier declares a code segment as a critical section with a given name. This is done with the `critical` *directive-type*. All the critical sections with the same name are mutually exclusive. This is accomplished through the use of the same FXCC lock for all of them. This directive implies a synchronization between processors to work on some shared data that may be modified. This directive can take the single optional clause `flushinval` to specify a list of variables. These variables are those that may be written in the critical section, or read in the critical section and written somewhere else. In this case, the lines with such variables are written back from the processor's cache (if dirty) and invalidated before entering the critical section. This forces the processor to read the latest version of the variables in the critical section. Moreover, these cache lines are written back to memory (if dirty) when the processor exits the section. This enables the next processor that will enter the critical section to access the new version of the variables. If the `flushinval` clause is not present, the operations described are performed on the whole cache.

4.2 Data Modifiers

We propose three directives of this kind, namely `alignable`, `page_aligned`, and `align`. The first two directives can precede the declaration of a C `struct` or `union` and instruct the compiler to pad the data structure to align it. Specifically, the **alignable** directive increases the size of the data structure until it becomes a power of two; the **page_aligned** directive increases the size until it becomes a multiple of the page size. Finally, the **align** directive aligns the dimensions of an array to page boundaries. This directive takes a syntax such as *array_name*(**[]**)*, where the number of square-bracket pairs specifies which dimension to align.

4.3 Executable Directives

CFlex has several executable directives that perform a variety of functions. As an example, the **barrier** directive implements a barrier for n processors using FXCC locks. As another example, the **flush** directive specifies a series of variables to write back to memory if the corresponding lines are dirty in the cache of the processor. This directive takes the optional **invalidate** clause, which additionally causes these lines (or a subset of them) to be invalidated after the potential writeback.

4.4 Examples

To gain more insight into the CFlex directives, we now show several simple examples of their use. A first example involves traversing a linked list and performing some processing on each of its nodes in parallel (Figure 3). The first directive in the figure generates a synchronous task in the PHost that will execute a whole loop. The purpose of this directive is to provide a context for synchronization of other tasks that will be spawned inside the loop. Inside each iteration of the **for** loop that constitutes the synchronous task generates one asynchronous task. The latter executes on the PArray whose local memory bank contains **p->data**.

```
#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.

Each one of these asynchronous tasks receives a privatized copy of the value of pointer **p** for the corresponding iteration, and processes a node from the list. Since these tasks are asynchronous, the PHost task continues to iterate the loop and spawn tasks until it reaches the end of the loop. Then, the synchronous task waits for all of the asynchronous tasks to complete. When they do, the synchronous task finally completes.

In this example, we have illustrated the general scheme used to parallelize a loop: declare the loop as a synchronous task and each of its iterations as an asynchronous one. Since loops are the most common source of parallelism, we have extended CFlex with a **pfor** clause that tells the compiler to break the loop following it into a series of asynchronous tasks and wait for their completion before continuing. Thus, the previous loop can be re-written using a **pfor** clause as shown in Figure 4. Note that although OpenMP is largely designed for loop parallelism, a parallel version of this loop in OpenMP would require the use of the **ordered** clause and would be less readable and efficient.

A more complex parallelization scheme is required when there are portions of code that must be run sometimes in the

```
#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.

PHost and sometimes in the PArrays. An example is shown in Figure 5. The example implements the routine that allocates the tree in the TreeAdd application from the Olden suite [15]. The routine allocates a binary tree of height **level**. Our parallelization strategy selects a level **cutlevel**. The nodes below that level are allocated by the PArrays and the nodes in that level and up to the root (highest level) are allocated by the PHost. Recall that the runtime system allows the PArrays to allocate and deallocate heap memory in parallel (Section 3.2).

```
tree_t *TreeAlloc (int level) {
  if (level == 0) return NULL;
  else {
    struct tree *new, *right, *left;

    new = (struct tree *) malloc(sizeof(tree_t));
#pragma FlexRAM phost if (level >= cutlevel)
    {
#pragma FlexRAM parray async if (level == cutlevel)
#pragma FlexRAM phost async else
      left = TreeAlloc(level-1);

#pragma FlexRAM parray async if (level == cutlevel)
      right=TreeAlloc(level-1);
    }
    new->val = 1;
    new->left = (struct tree *) left;
    new->right = (struct tree *) right;
    return new;
  }
}
```

Figure 5: Parallelized tree allocation.

The tree is allocated from root down. As long as the **level == cutlevel** condition is not satisfied, **TreeAlloc** is called by the PHost. Under these conditions, however, the PHost spawns a new task on the PHost to build the left subtree of each node. We follow the approach of creating multiple PHost tasks to avoid modifying the original code. When **cutlevel** is reached, both the left and right subtree allocation tasks are run on PArrays. Note that a single PArray task allocates a whole subtree. This is because a PArray ignores these execution modifiers since it cannot spawn new tasks. With this support, a whole subtree is allocated in a local memory bank. The resulting partitioning of the work is shown in Figure 6. In the figure, the dashed lines represent the spawn of a new task on the PHost. If there is not enough space in a memory bank to keep a whole subtree, some pages are allocated from another bank, and the PArray accesses them remotely. To avoid these remote accesses, it is best to choose a **cutlevel** that guarantees that a subtree built by a PArray fits in its local memory bank.

This work-partitioning strategy may be applied to any parallel processing of tree data structures. For example, the second step of the TreeAdd application is a reduction

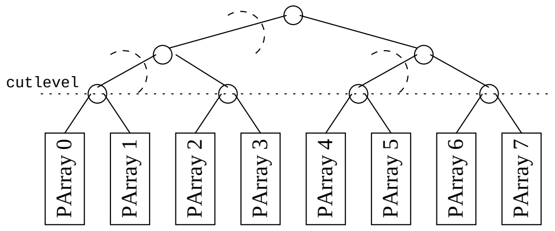

Figure 6: Partition of the work between the PHost (level `cutlevel` and above) and the PArrays for the code in Figure 5.

that adds the values stored in all the nodes of the tree; we have parallelized it in the same way[illegible] an additional optimization, the task that adds the valu[illegible] a subtree built by a PArray is spawned using the `on_home` clause to ensure that the reduction is performed by the PArray that owns the subtree. This approach exploits locality.

In pr[illegible]ve, our compiler generates two versions of the routine in [illegible]ure 5, one for the PHost and one for the PArrays. While the PHost version includes the directives, the PArray version does not. The reason is that PArrays cannot create new tasks, so these directives do not apply. Recall also that PHost and PArrays have different ISAs. Consequently, the backend compiler has to generate two versions anyway, even if the high-level code is exactly the same for both kinds of processors.

Finally, we consider an example where several data structures need to be processed together. In this case, while we can use the `on_home` clause to ensure that accesses to one data structure are local, the accesses to the other data structures may end up being remote. To address this problem, we could use a `migrate` clause to migrate the pages of the remote data structures, but the overhead could be high. Instead, an approach that often works is to make use of the first-touch page-placement policy of our OS. With this support, we may implicitly align at the page level the data structures that are used together throughout the code.

As an example, Figure 7 shows two lo[illegible] in the Swim application from the SPEC OMP2001 app[illegible]ion suite. In the f[illegible]e, the directives start slightly differently than before beca[illegible] the language used is FORTRAN. The first loop is from the `INITAL` routine and contains the first accesses in the program to vectors `UOLD`, `VOLD` and `POLD`. The `on_home` clause forces iteration `J` to be executed by the PArray that holds element `(1,J)` of matrix `U`. This PArray is also the first processor in the system to access column `J` of `UOLD`, `VOLD`, and `POLD`. As a result of our first-touch policy, the OS places the pages that contain such columns in the local memory bank of the same PArray. A similar strategy was used in previous loops to place the pages containing the columns of matrices `U`, `V`, and `P`. If the column sizes are (or can be made) such that the columns can be aligned to page boundaries, and each matrix starts at a page boundary, then all the accesses in the code end up being local. Figure 8 shows the resulting data layout assuming that each memory bank ends up allocating 8 columns from every matrix. In the figure, each box inside a memory bank represents a page-aligned chunk of memory that extends over several pages.

When related data structures are referenced later on by the same program, we have to distribute the corresponding

```
C$FlexRAM parray pfor on_home(U(1,J)) private(I)
      DO J=1, NP1
        DO I=1, MP1
          UOLD(I,J) = U(I,J)
          VOLD(I,J) = V(I,J)
          POLD(I,J) = P(I,J)
        END DO
      END DO
      ...

C$FlexRAM parray pfor on_home(U(1,J)) private(I)
      DO J=1,N
        DO I=1,M
          UOLD(I,J) = U(I,J) + ALPHA * ...
          VOLD(I,J) = V(I,J) + ALPHA * ...
          POLD(I,J) = P(I,J) + ALPHA * ...
          U(I,J) = UNEW(I,J)
          V(I,J) = VNEW(I,J)
          P(I,J) = PNEW(I,J)
        END DO
      END DO
```

Figure 7: Alignment of data structures using the `on_home` clause and the first-touch page-placement policy.

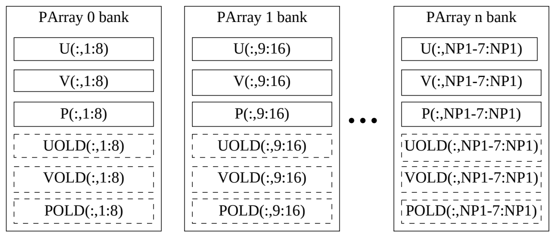

Figure 8: Alignment of the matrix columns from the first loop in Figure 7. In the figure, each box inside a memory bank represents a page-aligned chunk of memory that extends over several pages.

loop among the PArrays following the same policy. This is illustrated in the second loop of Figure 7, extracted from the `CALC3` routine.

Overall, note that none of the examples in this section required adding or modifying *any line* of the original sequential code. All the parallelization semantics have been expressed by means of compiler directives. This is typical of the codes that we have parallelized [illegible]FlexRAM.

5. INTELLIGENT MEMORY OPERATIONS

The family of compiler directives presented in the previous section enables the parallelization of a large set of codes and the exploitation of locality while hiding many of the system details from the programmer. However, to develop efficient programs, the programmer must be aware of the existence of the FlexRAM chips and must decide how to partition and coordinate the work between the PHost and the PArrays.

All these details can be hidden and performance can be improved with the use of library routines. Specifically, we envision the use of a library of Intelligent Memory Operations (IMOs). IMOs are me[illegible]y-intensive operations that can be encapsulated relativ[illegible]easily. IMOs perform common operations on data str[illegible]ctures that are often used in programs.

Function Description	Abstract Expression	Syntax
Apply func `f` with arg `a`	exec `f(v(i),a)`, $0 \leq$`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)`$\neq 0$, $0 \leq$`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)` $0 \leq$`i`$< N$	`v2=Vector_map(v,f,a)`
Reduce vector applying func `f`, whose neutral element is `a`	ret `f(...f(f(a,v(0)),v(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)` $0 \leq$`i`$< N$	`v3=Vector_map2(v,v2,f,a)`

Table 1: Examples of IMOs for vector containers.

Simple examples of IMOs are finding the minimum value in a vector of numbers or adding two matrices. Other, more structured examples of IMOs are STL classes [17]. Such IMOs may define and operate on containers such as vectors, lists, hash tables, or sets. They may make use of the intelligent memory to perform parallel allocations and deallocations, searches, insertions, retrievals, and other computations on the elements stored in these containers. Some examples of IMOs for vector containers are proposed in Table 1.

In practice, IMO libraries should implement two versions of the functions provided to the programmer: one to be used when no FlexRAM chips are detected in the system, and another one programmed in CFlex that exploits the capabilities of the FlexRAM chips. Both versions should be highly optimized and completely hide from the programmer issues such as task partitioning, scheduling, and synchronization.

6. APPLICATION TO OTHER ARCHITECTURES

There are other architectures that, like FlexRAM, are built out of a powerful PHost processor and a main memory with many simpler processors. Examples of such architectures are Active Pages [14] and DIVA [6]. For these architectures, CFlex and IMOs can also provide appropriate programming support. In fact, [14] illustrates the use of the STL array class to program Active Pages, very much in the line of our IMOs.

However, there are differences that seem to make Active Pages and DIVA more sensitive to the data placement than FlexRAM. In the case of Active Pages, the reason is that all the communications between the memory processors are serialized through the PHost. As a result, the PHost may become a bottleneck when many non-local accesses are required. Consequently, it would be important for a CFlex programmer to improve data placement and alignment of data structures.

In DIVA, exchange of data between memory chips requires the use of messages called parcels. Passing a parcel involves software processing by either user- or supervisor-level code at both ends [7], which makes it more expensive than the hardware approach followed by FlexRAM. Consequently, it would also be important for a CFlex programmer to maximize locality. Moreover, a CFlex compiler may improve performance by inserting efficient code required to manage the message passing when the communication between tasks is regular enough.

7. EVALUATION ENVIRONMENT

For our evaluation, we use an execution-driven simulation infrastucture that can model aggressive out-of-order superscalar processors and complete memory subsystems [12]. In the following, we describe the architecture modeled and the applications used.

7.1 Architecture Modeled

Our baseline architecture is a workstation with a high-performance 1.6 GHz five-issue PHost processor similar to IBM's Power4 [3]. The performance of this workstation is compared to that of two upgraded versions of it: one where the plain main memory is replaced by a single FlexRAM chip, and one where it is replaced by two FlexRAM chips.

The main architectural parameters of the system modeled are shown in Table 2. In the table, the times for each processor are measured in that processor's cycles. Note that the PHost is very powerful, has a large L2 cache, and is able to sustain many simultaneous memory accesses. Consequently, the baseline architecture is very aggressive. On the other hand, recent advances in merged logic-DRAM technology seem to enable the integration of high-speed logic with high-density memory in the same chip [9, 10]. Consequently, we have set the frequency of the PArrays to be 75% of that of the PHost. Recall that each FlexRAM chip has 64 PArrays. As shown in Table 2, these PArrays are much simpler than the PHost. Specifically, each PArray has a single integer adder and shares an integer multiplier with 3 other PArrays. Moreover, PArrays lack floating-point hardware, which they emulate in software. We assume that the latencies of emulating a floating-point add/subtract, multiply, and divide/sqrt operation are 3, 10, and 80 cycles, respectively.

We also simulate the parts of the OS and run-time system that are most likely to be exercised. For the OS, this includes building and keeping a two-level page table, allocating and mapping the virtual pages to the appropriate physical pages, maintaining the TLBs in both PHost and PArrays, and performing task scheduling. As for the run-time system, we model it completely, including task spawning, memory allocation by both the PHost and PArrays, and periodic polls and other accesses to synchronize PHost and PArrays. The size of the pages used is 16 Kbytes.

7.2 Applications Used

To evaluate the programmability and performance of a FlexRAM system using CFlex or IMOs, we select eight applications. These applications have a wide variety of characteristics, which can help us discover which attributes are

PHost Processor	PHost Caches	Bus & Memory
Freq: 1.6 GHz	L1 Size: 32 KB	Bus: Split Trans
Issue Width: 5	L1 OC,RT: 1,3	Bus Width: 8 B
Dyn Issue: yes	L1 Assoc: 2	Bus Freq: 400 MHz
I-Window Size: 64	L1 Line: 128 B	Mem RT: 180
Ld,St Units: 2,1	L1 MSHR: 16	(112.5 ns)
Int,FP Units: 5,4	L2 Size: 1 MB	
Ld Queue: 32	L2 OC,RT: 4,12	
St Queue: 32	L2 Assoc: 8	
BR Penalty: 12	L2 Line: 128 B	
TLB Entries: 128	L2 MSHR: 8	
PArray Processor	**PArray Cache**	**FlexRAM Torus, Bus**
Freq: 1.2GHz	L1 Size: 8 KB	Avg Torus RT: 14
Issue Width: 2	L1 OC,RT: 1,2	Torus Freq: 1.2 GHz
Dyn Issue: no	L1 Assoc: 2	Bus: Split Trans
Ld,St Units: 1,1	L1 Line: 32 B	Bus Width: 8 B
Int,FP Units: 1,0	Blocking	Bus Freq: 400 MHz
Ld,St Queue: 2,2		
BR Penalty: 6		
TLB Entries: 32		
PArrays/Chip: 64		

Table 2: Parameters of the architecture modeled. In the table, BR, OC, RT, and MSHR stand for branch, occupancy, latency of a round trip from the processor, and miss status handling register, respectively.Each PArray has a single integer adder and shares an integer multiplier with 3 other PArrays.

most suitable for FlexRAM, and what problems arise in FlexRAM programming. These applications have been annotated by hand with CFlex directives or [illegible]n calls to an IMO library that we created. [illegible] have a[illegible]modified the SUIF compiler [18] to accept C[illegible] pragmas and compile the applications. The resulting executable file is passed to our simulator infrastructure.

The IMO library contains operations to handle singly-linked lists. It includes both STL-like operations such as insertion, retrieval, or search, and many high-level operations[illegible]xamples of the latter include those in Table 1 applied[illegible]inked lists, and the processing with a function of all the pairs of elements taken from two lists. The library uses a linked list that is distributed amon[illegible]e PArrays. It makes extensive use of the run-time syst[illegible] functionallity that enables PArrays to allocate and deallocate portions of heap memory in parallel. All the operations are used in our applications. Overall, the library contains 714 lines of soure code, including the CFlex directives.

Table 3 lists the applications used. TSP and TreeAdd are taken from the Olden suite of pointer-intensive sequential applications [15], Swim and Mgrid are from the SPEC OMP2001 suite, Dmxdm and Spmxv are numerical kernels, and Distance and Pat[illegible]e written from problem descriptions in [4]. Distance a[illegible]Path are [illegible]d with IMO library calls, while the other applications u[illegible]Flex directives.

We use four axes to broadly classify the behavior of each application. Specifically, the access patterns may be irregular due to pointers (*Ptr*) or due indirections in the form of subscripted indices (*Ind*), or regular (*Reg*). The computation may use mostly integer (*Int*) or floating-point (*FP*) operations. When several data structures are involved in the computation, we are able to align all of them (*Yes*), only some of them (*Part*), or none of them (*No*). Finally, the typical number of instructions in the tasks sent to the PArrays may be tens of thousands (*Small*), hundreds of thousands (*Med*), or over one million (*Large*). The tasks in Mgrid have a variety of sizes. Overall, we can see that our applications cover a wide variety of behaviors.

The table also lists the data set size of the applications, and the average Instructions Per Cycle (IPC) of the applications running on the architecture without FlexRAM chips. The last three columns of the table attempt to estimate the effort required to map the applications to the FlexRAM system. [illegible]m left to right, they list the original number of lines of c[illegible], the number of CFlex directives inserted, and the additional lines of code required to map the code to the FlexRAM system. For the two applications coded with IMO calls, the last two columns have a slightly different meaning: number of CFlex directives in the IMO functions used, and the static number of calls to IMO functions, respectively. The original code size reflects the number of lines of the applications without including the IMO library. Note that [illegible]e effort has been made to optimize all the parallel ver[illegible]s, other than using a strategy that distributes the tasks evenly among the PArrays. Rather, we have stressed the simplicity of the mapping process by making as few modifications as possible, as the figures in the table show.

7.2.1 Details on Individual Applications

To help understand the mapping better, we now give some details on individual applications. We start with TSP and TreeAdd, which operate on trees built with pointers. TreeAdd is parallelized and mapped as discussed in Section 4.4. TSP follows a similar approach but has some differences. Specifically, subtrees in TreeAdd are processed independently by different PArrays, while the PHost performs the computation above a certain tree level. In TSP, instead, processing a node of the tree requires accessing the whole subtree below the node. Thus, assigning the processing of the upper levels of the tree to the PHost would leave a [illegible]f parallelism unexploited. In our parallelization, we let [illegible]rays work at all levels of the tree and only reserve the root for the Phost. As the processing moves up the tree, there are fewer subtrees, which means both fewer active PArrays and that those PArrays have to access data in more memory banks.

Swim and Mgrid use the *test* input data set. Their access patterns are very regular and floating-point operations dominate the computation. We exploit the vast loop-level parallelism in these applications by simply replacing the original OpenMP directives by the corresponding CFlex ones. Note that, although the data set size of Swim uses about 28 Mbytes, its page [illegible]print in memory is larger than 64 Mbytes because of in[illegible]al page fragmentaton. As a result, Swim requires at least two FlexRAM chips to execute without intensive swapping. Consequently, we [illegible]not perform experiments with Swim using a single Flex[illegible] chip.

Dmxdm and Spmxv are matrix multiplication kernels. Dmxdm multiplies two 1000×1000 dense matrices of double precision floating point elements with blocking in the three dimensions. Each submatrix is copied into consecutive locations to improve the locality. One of the loops is also unrolled and jammed. Spmxv multiplies a sparse 10000×10000 matrix with three million entries by a vector. The matrix is stored in Compressed Row Storage (CRS) format [1]. Column indices and non-zero values are not aligned because of their different size (four and eight bytes, respec-

Applic.	Coding	Application Characteristics				Data Set Size (MB)	Baseline IPC	Original Lines	Number Directives	Additional Lines
		Access	Data	Align	Task Size					
TSP	CFlex	Ptr	FP	-	Large	22	0.85	485	12	5
TreeAdd	CFlex	Ptr	Int	-	Large	40	1.01	71	8	4
Swim	CFlex	Reg	FP	Yes	Med	28	0.95	272	8	0
Mgrid	CFlex	Reg	FP	Yes	Var	55	2.35	470	13	0
Dmxdm	CFlex	Reg	FP	Part	Large	23	3.47	81	1	2
Spmxv	CFlex	Ind	FP	No	Med	36	0.59	47	1	0
Distance	IMOs	Ptr	Int	-	Large	1	2.04	108	17	7
Path	IMOs	Ptr	Int	-	Small	13	0.33	165	17	9
Average						27.3	1.45	212.4	9.6	3.4

Table 3: Characteristics of the applications used.

tively). These two kernels are parallelized by assigning a different PArray to compute a block of rows of the destination matrix (in Dmxdm) or a set of consecutive elements of the destination vector (in Spmxv). In both kernels, a single directive is required to parallelize the outer loop.

Finally, Distance and Path are programs with singly-linked list data structures that use our IMO library of operations on these structures. Distance takes a set of points in a two-dimensional space and finds all pairs of points that are closer than given distance; Path finds the shortest path between two given points in a graph. The IMO functions are designed to be very efficient for both the sequential (no FlexRAM) and the parallel (FlexRAM) execution of these applications. Still, there are some cases where the performance of the sequential execution can be hindered by IMO code structure that is better suited for parallel execution. In those cases, we [illegible]te versions of these IMO functions that are optimized for [illegible] sequential execution, and we use them when evaluating the no-FlexRAM architecture.

8. EVALUATION

8.1 Application Speedups

To evaluate the impact of the intelligent memo[illegible]e use the execution speedups of the FlexRAM system over the baseline workstation. We examine FlexRAM systems with one or two FlexRAM chips. Figure 9 shows the resulting speedups for each application and their geometric m[illegible] Recall that, to accommodate the working set of Swim, [illegible] need two FlexRAM chips. In the figure, the speedups correspond to the execution of the complete applications. All applications spend more than 99% of their original execution time in the section of the code parallelized with CFlex.

The figure shows that, for one FlexRAM chip, the speedup figures are quite good: they range from 1.5 to 40, with a geometric mean of 9. In general, the applications with the highest speedups are those with irregular access patterns and those with integer computation. This is li[illegible] because the other types of applications are relatively be[illegible]matched to the large caches and good floating-point support of the PHost-only baseline workstation. In addition, applications such as Path where PArray tasks largely use data located in the local memory bank obtain better speedups than applications such as Spmxv where PArrays require data from other banks.

The locality of PArray accesses also affects the changes in speedups as we go from one to two FlexRAM chips. In

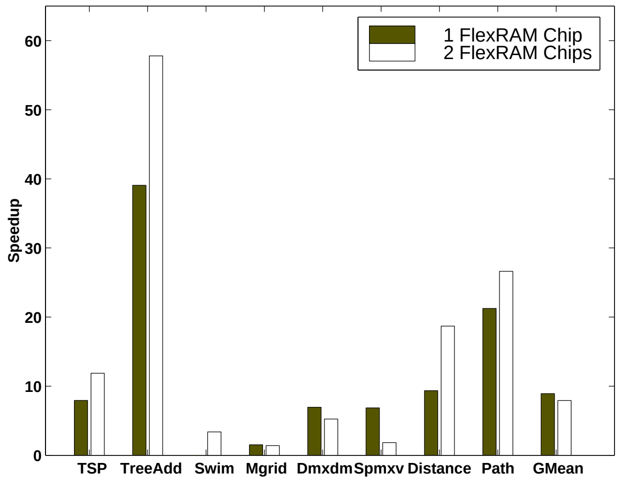

Figure 9: Execution speedups obtained using the FlexRAM system.

applications with good locality, the speedups go up, while in those with poor locality, the opposite occurs. In the second class of applications, the FlexRAM bus becomes a bottleneck for accesses to banks in other chips. Overall, without considering the contribution of Swim, the geometric mean of the speedups for two FlexRAM chips is also about 9. In general, contention in the FlexRAM bus and overheads due to synchronization and task spaw[illegible]ill grow with the number of FlexRAM chips. Conseq[illegible]ly, unless the application requires little data movement and synchronization, it is generally advisable to use the smallest number of FlexRAM chips required to hold its data set.

8.2 Compiler and Run-Time Optimizations

Our experiments produced some unexpected results. Specifically, note that the CFlex versions of our applications often use the `on_home` clause to [illegible]rage our first-touch page allocation policy and align dat[illegible]ructures for local computation. We call these versions *Opt*. The speedups shown in Figure 9 have been calculated using these versions. Surprisingly, we found that CFlex versions of Swim and Mgrid without any `on_home` clauses are faster than their corresponding *Opt* versions. In these new versions, which we call *NoOpt*, tasks are as[illegible]ed following the default scheme when the `on_home` cla[illegible]s not present: round ro[illegible]across chips and then, within a chip, round-robin across PArrays. Pages are still allocated using the first-touch policy. The difference in speedups between the *Opt* and *NoOpt* versions is shown in the first two bars of Figure 10.

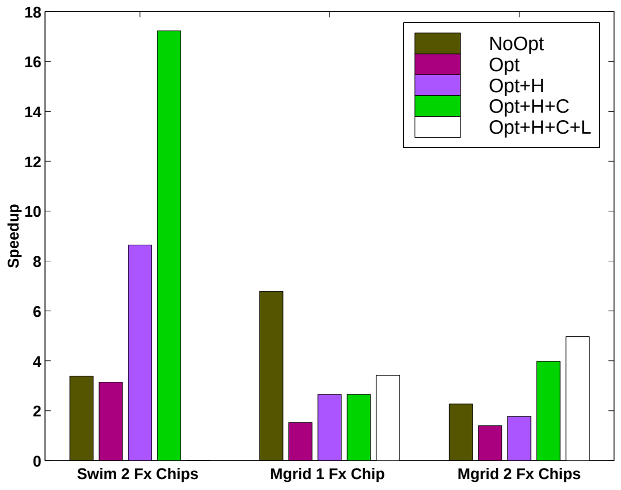

Figure 10: Impact of compiler and run-time optimizations for task spawning and mapping.

These experiments helped us identify at least three inefficiencies in the way tasks are spawned and mapped. The first inefficiency occurs in the way tasks are spawned under the `on_home` clause. Therefore, it only affects the *Opt* version. It occurs when the computation is to be assigned to consecutive tasks accessing consecutive pages in the memory bank of the same PArray. The original implementation of the compiler generates one task for each page and assigns all the tasks to the same PArray. Unfortunately, the creation of so many tasks causes significant spawning and synchronization overheads. Moreover, assigning them all to the same PArray may reduce parallelism. The reason is that our run-time system can only spawn a task when all the previous tasks generated by the PHost have already been spawned. Moreover, if the destination PArray is busy executing another task, the spawn request is queued up in a register of the chip's FXCC. If the FXCC runs out of registers, the spawn request cannot be queued and the run-time system has to wait for tasks to finish.

To eliminate this inefficiency, we changed our compiler as follows. When the consecutive tasks would access consecutive pages of the same memory bank, the compiler combines all the work into a single task. This approach eliminates overheads and the potential run-time stall problem mentioned above. We call this optimization *H* for *home-allocation*, and apply it to the *Opt* versions to obtain *Opt+H*. Figure 10 shows that *Opt+H* delivers higher speedups, especially for Swim.

The second inefficiency occurs in machines with more than one FlexRAM chip when tasks are mapped without the `on_home` clause. The default policy in this case, as stated above, is to map the tasks round robin among the FlexRAM chips for two reasons: to balance the usage of the chips and to reduce the likelihood of run-time stall due to running out of FXCC registers. Unfortunately, consecutive task spawns generate tasks that usually access related pieces of data, and often share the same data. Spawning these tasks on different chips often causes our first-touch page allocator to map the pages of consecutive portions of vectors and arrays on different chips. As a result, if tasks want to access information that is near in the virtual space, they are forced to use the FlexRAM bus and go across chips. This affects particularly the degree of locality that can be achieved in the *Opt+H* versions.

To eliminate this inefficiency, we change the mapping policy for consecutive tasks when the `on_home` clause is not present. We perform round-robin mapping of tasks within a chip before moving to mapping tasks to the next chip. Consequently, each chunk of 64 consecutive tasks is mapped in the same chip. We call this optimization *C* for *consecutive-on-chip*, and apply it to the *Opt+H* versions to obtain *Opt+H+C*. Figure 10 shows the resulting speedups, which are now significantly higher. Note that *Opt+H+C* does not apply to single-chip systems.

Finally, there is a third, potential inefficiency that is intrinsic to the use of the `on_home` clause. When assigning the computation to the PArray(s) on whose bank the data is located, we certainly obtain better locality. Unfortunately, we also restrict the number of PArrays that cooperate in the computation and, therefore, restrict parallelism.

Unfortunately, addressing this inefficiency involves distributing the data among as many PArrays as possible, which has negative effects on locality. Besides, we are limited by the fact that the granularity of the distribution in our system is the page. Therefore, computation to be parallelized using the `on_home` clause that operates on small pieces of data can only be split among a few PArrays. This would particularly hurt Mgrid's performance. Consequently, we optimize task spawning and mapping as follows: `on_home` clauses are applied only on loops that have more than 60 iterations. Otherwise, the loop is parallelized without applying the clause, thus losing locality but gaining in parallelism. This optimization attempts to ensure that the number of PArrays that execute the loop iterations is not too reduced despite the restriction that `on_home` imposes.

We call this optimization *L*, for *limited on_home*. The last bar in Figure 10 shows the improvement when applying this optimization to the *Opt+H+C* version of Mgrid. The other applications do not need this optimization.

8.3 Hardware Optimizations

We have examined other ways of improving the performance of our system. For example, applications that spawn many tasks would benefit from more intelligent FXCCs, particularly when the tasks have small size. The current system fills in the buffer with the information for each individual task and then communicates with a FXCC to attempt to spawn the task. Then the process in repeated for each new task. FXCCs could be improved to allow the runtime system to send a single request to a FXCC to spawn several tasks. The request would provide the number of tasks to spawn, the pointer to the code, and a vector of pointers to their buffers. Another possibility would be to locate the buffers in memory positions separated by a constant stride. In this case the address of the initial buffer and the stride would be communicated to the FXCC. In either case, the multispawn facility of the FXCC would reduce the overhead of spawning multiple tasks. The system could be further extended to allow the different tasks spawned with the same message to run different codes, but the scope of applicability of this

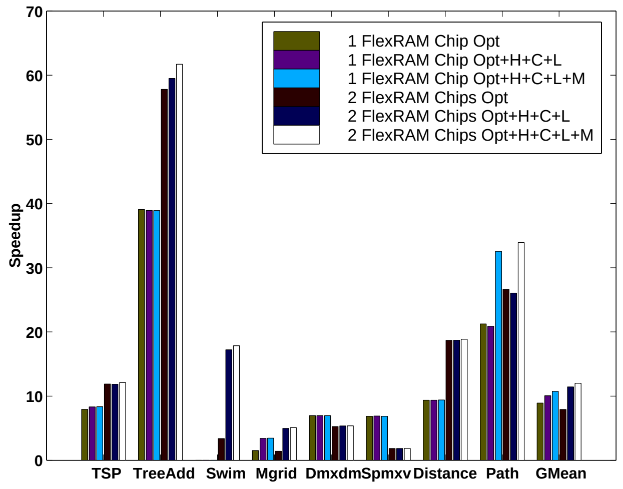

Figure 11: Effect of three optimizations in the system. Optimization `H` lies in generating one task per chunk of consecutive pages when the `on_home` clause is applied. Optimization *C* refers to the consecutive spawn of consecutive tasks in the same chip. Optimization *L* limites the usage of the `on_home` clause to loops with a minimum number of iterations. Intelligent multispawn FXCCs are represented by *M*.

enhancement is smaller. We call this optimization *M* for *Multi-spawn*, and apply it to the *Opt+H+C* versions to obtain *Opt+H+C+M*. Fig. 11 shows how the initial speedups in Fig. 9 evolve for one and two chips applying the possible optimizations we have discussed so far. The effect of optimization *H* is not shown separately because it only affects Swim and Mgrid, so Fig. 10 suffices for this purpose. Optimization *L* is only applicable to Mgrid, so the same applies. Multispawn FXCCs improve the behavior of virtually all of the applications. The effect is particularly noticeable in Path, which needs to spawn many small tasks.

8.4 Other Optimizations

Finally, highly optimized versions of the codes can be written. For example, we have developed a complex Dmxdm code version with 148 lines of code and 12 CFlex directives. This code chooses a leader PArray in each FlexRAM chip to perform the movements of data which may come from another FlexRAM chip. The other PArrays in the chip synchronize with it using FXCC locks in order to copy the data from its buffer once it has finished. The resulting code is about 5% slower than the original one when only one FlexRAM chip is available, but performance is doubled when two chips are used. Sometimes the improvements obtained through more complex implementations turn out to be smaller than the new spawn and synchronization costs they require. This happened in our experiments with Spmxv, for example.

No quatitative comparison with other intelligent memory architectures and programming environments is provided because of the enormous differences both in the architectural assumptions and the programming paradigms.

9. RELATED WORK

We now compare CFlex with OpenMP [13] and HPF [8], the most widely known parallelization directives. CFlex is inspired by the former, so their approaches have many points in common, such as the explicit use of threads. There are, however, several important differences with OpenMP. One is that the OpenMP machine model is UMA and, as a result OpenMP lacks the locality related clauses found in CFlex. Local memories are meaningful to HPF but it uses replication and alignment to take advantage of them. While this strategy is adequate for regular data structures, data structures enabled by C pointers and structs, which are the focus for our work, cannot be partitioned with these directives. CFlex provides mechanisms for implicitly distributing and even aligning both regular and irregular data structures, as the examples in Sect. 4.4 have shown.

Task definition and synchronization is also more powerful in CFlex than in OpenMP and HPF. The `sync`/`async` clauses enable the spawn of new tasks dynamically outside loops. In this way it is possible to parallelize recursive algorithms and the processing of lists, trees and other pointer-based structures using our family of directives. As we explained before, this is particularly interesting for FlexRAM, as PIM architectures are particularly well suited for codes with irregular access patterns. This gives CFlex a very important advantage over OpenMP, which can only parallelize iterative constructs of the `for`/`do` type and the statically nested parallelism of the `sections` and `section` directives. HPF is also primarily designed to exploit loop level parallelism, although version 2.0 [8] includes the `TASK_REGION` directive, which allows to implement parallel sections, nested parallelism and data parallel pipelines. Still, it is less powerful than CFlex, as the generated tasks must honor several restrictions. For example all the data they access must be mapped (local) to the active processor subset.

Finally, both in OpenMP and HPF all processors have the same capabilities. But CFlex's orientation to intelligent memory systems forces it to explicitly distinguish two very different kinds of processors: the main processors(s) of the system, PHost(s), and the memory processors, PArrays.

10. CONCLUSIONS

A programming environment oriented to intelligent memory architectures has been presented and evaluated. The target architecture is a variation of the FlexRAM architecture presented in [11], but it is suitable to other architectures that use the PIM chips in a similar way. Programming is based on a set of compiler directives inspired by OpenMP called CFlex. Our directives allow the parallelization of a much larger class of codes than OpenMP and they include clauses that allow the programmer to make a better exploitation of locality. The programmer does not need to know any detail of the system to use these directives, but he is in charge of specifying the partitioning of work between the PHost and the PArrays and how the generated tasks must synchronize. These directives can also be used to program general NUMA systems, as they allow to parallelize codes without referring to PIMs. An alternative or complementary programming may be developed using libraries with highly optimized functions that use the FlexRAM chips while hiding completely their existence. We call these functions Intelligent Memory Operations. The programming environment is completed by some Operating System extensions, and an user runtime system.

A selection of benchmarks with very different properties and behaviors was chosen to evaluate our system. CFlex proved to be a flexible language that allowed to parallelize the codes, sometimes following relatively complex parallelization schemes, making minimal changes. Despite choosing an aggressive baseline, high speedups of ten or even more were obtained for applications with irregular access patterns, mainly on codes based on integers. Floating point operations prevent great performance improvements because of the simplicity of our processors in memory. The experiments show the importance of data alignment and locality to obtain the best results. Some optimizations were suggested by the experiments that raised the geometric mean of the speedups obtained using one and two FlexRAM chips from 9 to 10.7, and from 7.9 to 12, respectively. Still, further improvements may be achieved by careful optimization of the codes, which indicates that the usage of IMOs is probably the best option for most users.

11. ACKNOWLEDGMENTS

We thank the referees and the members of the Polaris and IACOMA groups for their valuable feedback, particularly George Almasi.

12. REFERENCES

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Press, 1994.

[2] J. Chame, J. Shin, and M. Hall. Code Transformations for Exploiting Bandwidth in PIM-Based Systems. In *Solving the Memory Wall Problem Workshop*, June 2000.

[3] K. Diefendorff. Power4 Focuses on Memory Bandwith. *Microprocessor Report*, 13(13), October 1999.

[4] C. Foster. *Content Addressable Parallel Processors.* Van Nostrand Reinhold Co, New York, 1976.

[5] B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. CFlex, a Programming Language for the FlexRAM Intelligent Memory Architecture. Technical Report UIUCDCS-R-2002-2287, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2002.

[6] M. Hall et al. Mapping Irregular Aplications to DIVA, a PIM-based Data-Intensive Architecture. In *Supercomputing*, November 1999.

[7] M. Hall and C. Steele. Memory Management in PIM-Based Systems. In *Proceedings of the Workshop on Intelligent Memory Systems, held in conjunction with Architectural Support for Programming Languages and Operating Systems*, November 2000.

[8] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. 1997.

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

[10] S. S. Iyer and H. L. Kalter. Embedded DRAM technology: opportunities and challenges. *IEEE Spectrum*, 36(4):56–64, April 1999.

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

[12] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[13] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface Version 2.0. March 2002.

[14] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *International Symposium on Computer Architecture*, pages 192–203, June 1998.

[15] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[16] Y. Solihin, J. Lee, and J. Torrellas. Automatic Code Mapping on an Intelligent Memory Architecture. *IEEE Transactions on Computers*, 50(11):1248–1266, November 2001.

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

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

[19] S.-M. Yoo, J. Renau, M. Huang, and J. Torrellas. FlexRAM Architecture Design Parameters. Technical Report CSRD-1584, Department of Computer Science, University of Illinois at Urbana-Champaign, October 2000. http://iacoma.cs.uiuc.edu/flexram/publications.html.