Automatically Mapping Code on an Intelligent Memory Architecture*

Jaejin Lee[‡], **Yan Solihin**[†§], and **Josep Torrellas**[†]
[†]University of Illinois at Urbana-Champaign
[‡]Michigan State University
[§]Los Alamos National Laboratory
http://iacoma.cs.uiuc.edu/flexram

ABSTRACT

This paper presents an algorithm to automatically map code on a generic intelligent memory system that consists of a host processor and a simpler memory processor. To achieve high performance with this type of architecture, code needs to be partitioned and scheduled such that each section is assigned to the processor on which it runs most efficiently. In addition, the two processors should overlap their execution as much as possible.

With our algorithm, applications are mapped fully automatically using both static and dynamic information. Using a set of standard applications and a simulated architecture, we show average speedups of 1.7 for numerical applications and 1.2 for non-numerical applications over a single host with plain memory. The speedups are very close and often higher than ideal speedups on a more expensive multiprocessor system composed of two identical host processors. Our work shows that heterogeneity can be cost-effectively exploited and represents one step toward effectively mapping code on intelligent memory systems.

1 INTRODUCTION

Integrating substantial processing power and a sizable memory on a single chip can potentially deliver high performance by enabling low-latency and high-bandwidth communication between processor and memory. This type of architecture, which is popularly known as intelligent memory or processor in memory, has been recently proposed for many systems [8, 12, 13, 14, 16, 19, 22, 24].

Some proposals use this architecture for the main processing unit in the system. Examples of such systems are IRAM [14], Shamrock [13], Raw [24], and Smart Memories [16] among others. Other proposals, instead, use this architecture for the memory system, replacing plain memory chips. In this case, intelligent memory chips act as co-processors in memory that execute code when signaled by the host (main) processor. Examples of proposed systems that use this approach are Active Pages [19], DIVA [8], and FlexRAM [12].

In this second class of systems, we have a heterogeneous mix of processors: host and memory processors. A host processor is more powerful, is backed up by a deep cache hierarchy, and suffers a high latency to access memory. A memory processor is typically less powerful, has a lower memory latency and, at least in theory, is significantly cheaper. The question that we address in this paper is: how do we automatically program these systems?

In previous work on these systems [5, 8, 12, 19], the programmer is expected to identify and isolate the code sections to run on the memory processors. This process is time consuming and error prone. Furthermore, in our experience, visual inspection of the code may not reveal much about which processor is best at running a given code section. In addition, previous work has largely focused on executing sections of code on only a set of identical memory processors. This approach is often not much different from running code on a parallel processor.

Our goal, instead, is to automatically partition the code into homogeneous sections and then schedule each section on its most suitable processor, while maximizing host and memory execution overlap. No knowledge of the code should be assumed from the user.

To this end, this paper presents an algorithm embedded in a real compiler that automatically maps code to a system with both host and memory processing. To simplify the analysis, we only generate code for an architecture with a single host and a single memory processor. Using a set of standard applications and a software simulator for the architecture, we show average speedups of 1.7 for numerical applications and 1.2 for non-numerical applications over a single host with plain memory. The speedups are similar and often higher than *ideal* speedups on a more expensive multiprocessor system composed of two identical host processors. Overall, our work shows that heterogeneity can be cost-effectively exploited in an automated manner. It represents one step toward effectively mapping code on intelligent memory systems.

The rest of the paper is organized as follows: Section 2 overviews the intelligent memory architecture used; Section 3 presents our algorithm; Section 4 describes the evaluation environment; Section 5 evaluates the algorithm; and Section 6 discusses related work.

2 INTELLIGENT MEMORY SYSTEM

For this work, we assume a server with a memory system enhanced with processing power. The machine has two types of processors: the off-the-shelf processors that come with ordinary servers (*P.hosts*) and the processors in the memory system (*P.mems*). To simplify the analysis, in this work we use a simpler architecture with only one processor of each type (Figure 1-(a)). Supporting multiple processors of each type requires extending the techniques that we will present, possibly by augmenting them with conventional parallelization techniques.

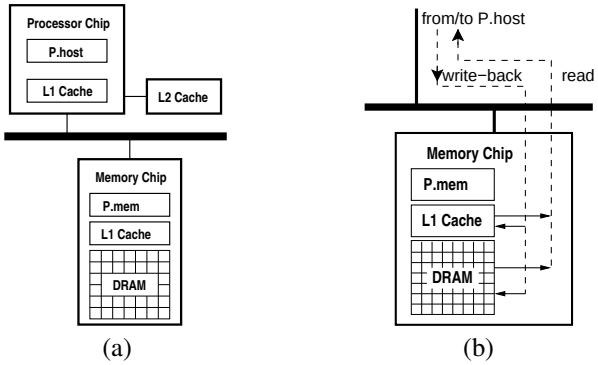

Figure 1: Intelligent memory architecture considered.

*This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, Michigan State University, and gifts from IBM and Intel.

Typically, P.host is a wide-issue superscalar with a deep cache hierarchy and a high memory access latency, while P.mem is a simple, narrow-issue superscalar with only a small cache and a low memory access latency. Consequently, to run efficiently, computing-bound code sections should be run on P.host, while memory-bound sections should run on P.mem.

To reduce the cost of the system, a restriction that we impose is that the chips for the host and memory processors are connected with an off-the-shelf interconnection. As a result, only P.host can be the master of the interconnection and initiate transactions; P.mem cannot initiate interconnection transactions. In addition, there is no hardware support to ensure coherence between the P.host and P.mem caches.

We assume, however, some simple support in P.mem's cache that makes programming easier (Figure 1-(b)). Specifically, when P.host writes back a line to memory, P.mem's cache is automatically updated if it contains a copy of the line. In addition, when P.host requests a line from the memory, P.mem's cache overwrites the returning data if it has a copy of the line.

For P.host and P.mem to execute an application concurrently, we need to support synchronization and data coherence correctly. For synchronization, since P.mem cannot be master of the interconnection, the most inexpensive scheme is to poll a special, uncachable memory location. Whichever processor arrives first sets the location and keeps polling until the other processor arrives. Depending on how sophisticated the memory controller is, the controller can offload the spinning from the P.host. Ensuring data coherence is harder. We consider it next.

2.1 Data Coherence

Since some sections of the code execute on P.host while others on P.mem, the data in the caches will become incoherent in the course of execution. To avoid incorrect execution, we must ensure that when a processor accesses a variable, it gets the latest version of it. This can be ensured in different ways, depending on the support available. For this work, we assume very simple support, namely that P.host can issue *write-back* and *invalidation* commands to control its caches. We use this support as follows:

1. Before P.mem starts executing a section of code, P.host writes back to memory all the dirty lines in its caches that P.mem may read or only-partially modify in that section. The partial-modification condition is necessary in case the two processors write to different words of the same line. Recall that, as a line is written back, it updates P.mem's cache if the latter has an old copy of the line. This support ensures that P.mem sees the latest versions of data.
2. Before P.host starts executing a section of code, P.host invalidates from its caches all the lines that P.mem may have updated in the previous section. This support ensures that P.host sees the latest versions of data: if P.host re-references lines written by P.mem, it will miss in its cache.

Therefore, in the general case, transferring execution from P.host to P.mem and then back to P.host induces three overheads on P.host: writing back some cache lines to memory, invalidating some cache lines and, later on, potentially missing in the caches to reload the invalidated lines.

In reality, the cost of these operations heavily depends on the quality of the compiler. The compiler can minimize the cost by performing careful dependence analysis to write back and invalidate only the strictly-necessary cache lines. It can schedule the write-backs in advance and in a gradual manner, to avoid bunching them up right before execution is transferred to P.mem. It can schedule the invalidations when P.host is idle waiting for P.mem. Finally, it can also insert prefetches to reload the data in P.host's caches in advance, but only when it is safe to do so.

3 AN ALGORITHM TO MAP THE CODE

We have implemented a compiler and run-time algorithm that automatically maps applications to the intelligent memory architecture of Section 2. The algorithm maps both numerical and non-numerical applications. For the numerical applications, the algorithm is embedded in the Polaris parallelizing compiler [3].

The algorithm has several parts. First, the code is partitioned into modules that have a homogeneous memory and computing behavior (Section 3.1). Then, using a static performance model or code profiling, it estimates which processor should each module run on (Section 3.2). Since static scheduling decisions may be poor, the algorithm also inserts code that identifies at run time where each module should run (Section 3.3). Finally, the algorithm enables the overlap of P.host and P.mem execution (Section 3.5).

3.1 Code Partitioning

The first step in the algorithm is to partition the code into *modules* or sections of code that have a homogeneous computing and memory behavior. In addition, a module should have good locality and be easy to extract from the code. We use two partitioning algorithms: *basic* partitioning and the more aggressive *advanced* partitioning. Basic partitioning finds *basic* modules, while advanced partitioning combines them into *compound* modules.

3.1.1 Basic Partitioning

Intuitively, the desirable characteristics listed above are most likely to be found in loop nests. Consequently, we define a *basic* module to be a loop nest where each nesting level has only one loop and possibly several statements. Such a loop nest may span several subroutine levels.

To identify basic modules, the algorithm starts off by marking as initial modules all innermost loops in the application that contain neither subroutine calls leading to loops nor *if* statements enclosing loops. Then, it tries to expand these initial modules, possibly crossing subroutine boundaries in the process.

For the expansion process, we first order the subroutines in the application starting from the leaves in the static call graph and working bottom up in the graph. We then consider one subroutine at a time. For a given subroutine, we order the modules depth first. Then, for each module M in the subroutine, we repeatedly apply the following two steps in sequence until the module stops expanding:

1. Given a statement s in the subroutine, which is neither a module, nor a subroutine call leading to a loop, nor an *if* statement enclosing a loop. If s dominates M and M postdominates s[1], and moving s immediately before M does not violate data dependences, we do so and merge s into M. If s postdominates M and M dominates s[1], and moving s immediately after M does not violate data dependences, we do so and merge s into M.
2. If M is the body of a *do* or *while* loop, then the loop becomes the new module. If M is the full body of the subroutine, then any call statements to the subroutine become the new modules.

After all the modules in the subroutine have been processed, we move on to the next subroutine. After all the subroutines have been processed, we need to perform one final operation. We examine each of the resulting modules. If a module is not a loop nest, we peel-off statements until it becomes a loop nest. After all these steps, we have all the basic modules.

As an example, the code in Figure 2-(a) contains two loops that are marked as initial modules, namely L1 and L2. After applying the algorithm described, the resulting basic modules are loops L3 and L2 as shown in Figure 2-(b).

[1]This condition implies that M is executed whenever s is executed and vice versa.

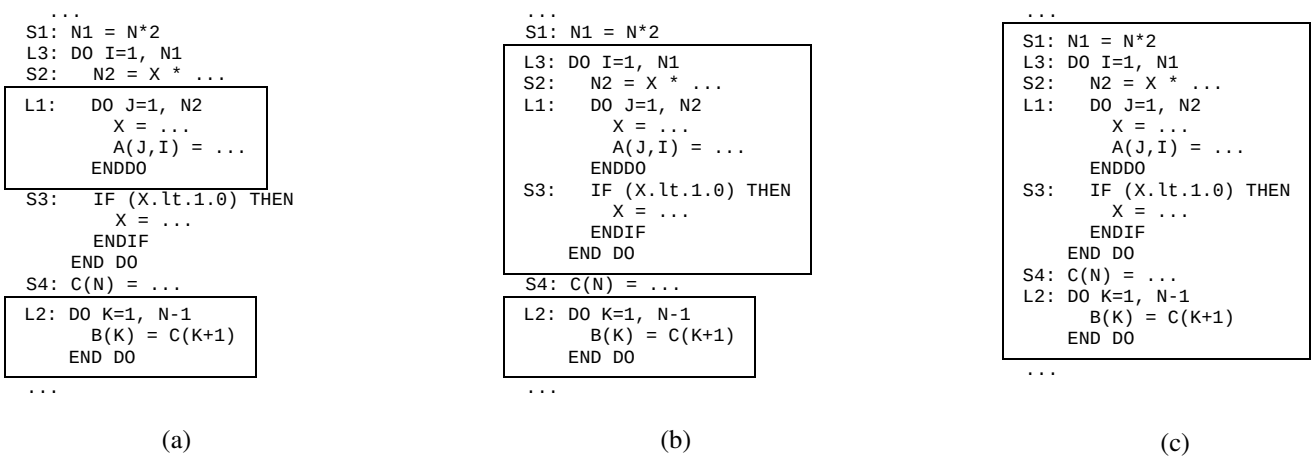

Figure 2: A sample code with two loops marked as initial modules (a) and its structure after basic partitioning (b) and advanced partitioning (c). We assume that the two modules in (b) have the same estimated affinity.

3.1.2 Advanced Partitioning

While basic modules may be fairly homogeneous, they may also be small. If so, it is possible that, relative to their small grain size, they induce unacceptably high overheads to keep the data in the caches coherent (Section 2.1) and to bundle up the code into units ready to execute (Section 4). With advanced partitioning, we try to increase the grain size of the modules and, therefore, reduce their relative overheads, at the possible expense of lowering their homogeneity.

In this algorithm, we generate *compound* modules out of basic modules. Compound modules do not have to be loop nests. They are generated by merging basic modules with nearby statements and other basic modules whose affinity is expected to be the same. We say that a module has *affinity* for P.host or P.mem if it runs faster on P.host or P.mem, respectively. The affinity of a module is estimated using a static performance model or code profiling (Section 3.2).

The advanced partitioning algorithm starts off by ordering the subroutines in the application starting from the leaves in the static call graph and working bottom up in the graph. Within each subroutine, the basic modules identified in Section 3.1.1 are marked and ordered depth first. The algorithm then works on one subroutine at a time. For a given subroutine, it applies an expansion sub-algorithm to every module in sequence. Then it applies a combining sub-algorithm to every module in sequence. The two sub-algorithms are repeatedly applied until the modules in the subroutine do not change further.

The expansion sub-algorithm expands a module. It works like the expansion in basic partitioning. Given a module M in a subroutine, it repeatedly executes the following steps until the module stops expanding:

1. Step 1 from Section 3.1.1.
2. Step 2 from Section 3.1.1.
3. If M is the *then* or the *else* clause of an *if* statement and the other clause of the *if* statement has no modules, the *if* statement becomes the new module. This step was not included in basic partitioning because it can create modules that behave very differently across invocations.

The combining sub-algorithm combines modules within a subroutine. Given two modules M_1 and M_2 in the subroutine that have the same estimated affinity, it executes the following steps:

1. If M_1 dominates M_2 and M_2 postdominates M_1: if we would not violate any data or control dependence by moving all the statements in control flow paths between M_1 and M_2, to immediately before M_1 or after M_2, we perform the move and combine the two modules into a single one.
2. If M_1 and M_2 are the *then* and *else* clauses respectively of an *if* statement: the *if* statement becomes the new module.

After a subroutine has been processed, we move on to the next one. After all the subroutines have been processed, we obtain the compound modules.

As an example, Figure 2-(c) shows the result of applying advanced partitioning to the code of Figure 2-(b). If we assume that the two basic modules have the same estimated affinity, the whole code becomes a compound module.

Note that there are modules so small that the overheads that they are expected to induce may overwhelm their execution time. We will see in Section 3.2 that these modules are statically assigned *undefined* affinity. If, in the advanced partitioning algorithm, one of these small modules is considered for combination with a large one, the algorithm assumes that the small module has the same estimated affinity as the large one. Consequently, it combines them. If two small modules are considered for combination, they are also combined but, if the resulting module is large enough, its estimated affinity may now become P.host or P.mem.

3.1.3 Advanced Partitioning with Retraction

One possible problem with the compound modules generated by advanced partitioning is that they may be very large and, as a result, may be invoked very few times during program execution. If this occurs, run-time adaptation (Section 3.3) is harder to apply because the module may not run enough times for the system to learn.

To address this problem, we also propose a variation of advanced partitioning that includes a retraction step at the end. Specifically, after all the compound modules have been identified, the algorithm selects those that (possibly through profiling) are expected to be invoked only 1-2 times. For each of these modules, the algorithm then starts peeling-off statements until it reaches an all-enclosing loop or a set of disjoint loops. In any case, the bodies of these loops or loop are the new compound modules. This is because these bodies, which are usually much larger than basic modules, are likely to be executed several times. With this approach, we allow run-time adaptation to work, at the expense of reducing the grain size of the module.

3.2 Affinity Estimation

To be able to combine modules under advanced partitioning, our algorithm must have the ability to estimate the affinity of individual modules. Such ability is also needed to decide where to schedule the execution of modules in case we use a static scheduling policy.

To estimate the affinity of modules, we use a static performance model. Currently, the model is designed only for numerical appli-

cations. For non-numerical ones, we use information gathered from two profiling runs of the code, one on P.host and one on P.mem, using a different input set than for the production run. The profiling runs measure the execution time and number of invocations of each module. In the rest of this section, we describe the static performance model.

3.2.1 Static Performance Model

The model is based on Delphi's performance predictor [4]. It estimates the execution time of a module on both P.host (T_{phost}) and P.mem (T_{pmem}), and selects the processor with the lowest time as the estimated affinity of the module. The model works by estimating the two major components of the execution time: computing time (T_{comp}) and memory stall time ($T_{memstall}$).

To estimate T_{comp}, we proceed as follows. Each instruction in the module is multiplied by its execution latency. The results are combined and grouped into three execution times: time for integer execution (T_{int}), floating-point execution (T_{fp}), and load/store execution (T_{ldst}). Next, we assume that one of these three types of instructions will determine the T_{comp} of the module by keeping its functional units busy all the computing time. Specifically, suppose that we have N_{int}, N_{fp}, and N_{ldst} integer, floating-point, and load/store functional units, respectively. Then, we estimate T_{comp} as:

$$T_{comp} = max(\frac{T_{int}}{N_{int}}, \frac{T_{fp}}{N_{fp}}, \frac{T_{ldst}}{N_{ldst}}) + T_{other} \quad (1)$$

In the formula, T_{other} is the estimated latency of special items like calls to libraries that compute square roots or intrinsic functions.

To estimate $T_{memstall}$, we estimate, for each level of the cache hierarchy, the number of cache misses in the module ($miss$) and the average processor stall time per miss ($stall$). Then, $T_{memstall}$ for a cache hierarchy is estimated as:

$$T_{memstall} = \sum_{i \in caches} miss_i \times stall_i \quad (2)$$

The number of cache misses in the module is estimated by using the data dependence structure of the code to predict the access patterns. The latter then drive a stack-distance model for the cache [4, 17]. The stack-distance model assumes fully-associative caches and, therefore, underestimates the number of misses. As for the average stall time per miss, it is hard to estimate since part of the cache miss latency is overlapped with computation. In addition, resource contention increases the stall. For simplicity, in our calculations, we use the full non-overlapped cache miss penalty without contention.

3.2.2 Affinity and Inaccuracy Window

All these approximations induce some error to the values estimated for T_{phost} and T_{pmem}. In addition, there are other sources of error that arise from the fact that some information may not be available at compile time. Examples of such information are the outcomes of branches and full data dependence structure information. Overall, however, the model delivers acceptable results. Recall that its goal is not to estimate the actual execution time as accurately as possible, but to estimate which of T_{phost} and T_{pmem} is smaller.

To be on the safe side, we use an *inaccuracy window* for the model. We assume that the error can be reasonably bounded by $err\%$ of the measured value and, therefore, have an inaccuracy window of $\pm err\%$. Consequently, our model only reports the affinity for a module if the inaccuracy windows $[T_{phost}(1 - err/100), T_{phost}(1 + err/100)]$ and $[T_{pmem}(1 - err/100), T_{pmem}(1 + err/100)]$ do not overlap. Otherwise, the affinity is reported as *undefined* (Table 1).

Finally, as shown in Table 1, very small modules are also assigned *undefined* affinity. The reason is that the overheads required to keep the data in the caches coherent (Section 2.1) and to bundle up the code into a module ready to execute (Section 4) are likely to overwhelm the execution time of one of these modules. By setting the affinity to *undefined*, we increase the chances that the very small module combines with nearby modules. In addition, for very small module sizes, the model becomes less accurate.

Condition	Affinity
$T_{phost}(1 + err/100) < T_{pmem}(1 - err/100)$	P.host
$T_{phost}(1 - err/100) > T_{pmem}(1 + err/100)$	P.mem
Windows intersect or module is very small	Undefined

Table 1: Estimating the affinity of a module.

3.3 Adaptive Execution

The algorithm can use the estimated affinity of the modules to individually schedule each module on either P.host or P.mem statically. Alternatively, the schedule can be decided *dynamically*. In this case, the compiler generates both P.host and P.mem versions of the module. In addition, it includes code in the module to measure at run time the execution time of some of its invocations or (if applicable) some of its loop iterations. These invocations or iterations where execution times are gathered are called *decision runs*. Based on the measurements in the decision runs, the run-time system determines the affinity of the module and schedules subsequent invocations or loop iterations of the module in the program appropriately.

Depending on the granularity of the code executed in a decision run, we propose *coarse*- and *fine*-grain dynamic scheduling strategies. In coarse strategies, the granularity is an entire module invocation, while in fine strategies, the granularity is a single iteration of the outermost loop. Of course, fine strategies are only applicable to basic modules and to compound modules that have an all-enclosing loop.

We propose two different coarse strategies: *coarse basic* and *coarse most recent*. In *coarse basic*, the module is executed and timed on one processor when it is first invoked in the program, and on the other processor in its second invocation. The affinity of the module is then determined by comparing the two measurements. The module is then executed for the remaining invocations in the program on the processor for which it has affinity.

In *coarse most recent*, the module is also executed first on one processor and then on the other. However, the scheduling for the remaining invocations in the program is not fixed at that point. Instead, every time that the module executes on a processor, we time it and compare the execution time to its most recent execution time on the *other* processor. If the latter is lower, we change the affinity of the module.

We propose two different fine strategies: *fine basic* and *fine first invocation*. In *fine basic*, in each invocation of the module, the first iteration of the loop is executed and timed on one processor, and the second one on the other processor. Based on these two measurements, the affinity is determined and fixed for the rest of the loop execution. The whole process is repeated in every invocation. In *fine first invocation*, the decision runs are performed only in the first invocation of the module. Once the affinity is determined after the second iteration, it is fixed for all subsequent iterations and invocations of the module.

Table 2 helps compare the different dynamic scheduling strategies. The table classifies modules into 8 different classes, based on the number of times the module is invoked in the program, and whether or not the module's behavior varies across invocations and across iterations of the same invocation. The table then lists the best strategy or strategies for each class of module.

Coarse strategies tend to be more effective for modules that are invoked relatively more times. The reason is that, in general, in the first two invocations of a given module, coarse strategies schedule the module on the wrong processor once. As a result, they are not

Num. Invocations	Behavior Across Invocations	Behavior Across Iterations	Best Strategy
> 2	Constant	Constant	Fine first invocation
	Constant	Variable	Coarse basic
	Variable	Constant	Coarse most recent, Fine basic
	Variable	Variable	Coarse most recent
1 or 2	Constant	Constant	Fine first invocation
	Constant	Variable	—
	Variable	Constant	Fine basic
	Variable	Variable	—

Table 2: Comparing the different dynamic scheduling strategies.

competitive for modules invoked very few times.

Fine strategies, on the other hand, are much less affected by the number of invocations. However, they assume that all the iterations in the loop behave similarly. Such an assumption is not accurate in many loops, particularly triangular ones. Consequently, fine strategies are not competitive for modules with variable iteration behavior.

Among coarse strategies, *coarse basic* only works well if the behavior of the module does not vary across invocations. *Coarse most recent*, instead, can adapt to changes in the behavior of the module across invocations. It uses a module's recent past to predict its future behavior. Consequently, if the module changes gradually, this strategy works well. However, sudden changes may cause this strategy to work poorly.

Of all the strategies, *fine basic* has the highest overhead, since execution transfer between P.host and P.mem happens at least once for every invocation of the module. However, it adapts to changes across invocations. Compared to all other strategies, *fine first invocation* has the lowest overhead. However, it is only useful when the behavior of the loop does not change across iterations or invocations.

Overall, we see from the table that no single strategy is best in all cases. In practice, it may be better to focus on the modules that are invoked many times, since they are more likely to contribute significantly to the overall execution time of the application. Moreover, it may be safer to assume that the behavior of a module will vary across both iterations and invocations.

Finally, our algorithm uses several heuristics. Modules with *undefined* affinity are always run on P.host. This approach is likely to reduce the overheads associated with transferring execution between processors. For a module with defined affinity, the first decision run is always scheduled on the processor for which the module has affinity. This approach minimizes the chances of executing the module on the wrong processor.

3.4 Parallelization

At this point, if our intelligent memory architecture included several P.mem processors, we could parallelize the modules assigned to memory. Similarly, if the architecture had several P.hosts, we could parallelize the modules assigned to the host. To do so, we could use many conventional compiler techniques. While such an approach is certainly interesting, we prefer to focus on an architecture with a single P.mem and a single P.host, and examine the less explored issue of overlapping execution between the two. We recognize, however, that both axes of parallelism affect each other and that both need to be included in a complete compiler algorithm for intelligent memory architectures.

3.5 Overlapped Execution

To further speed-up execution of the application, the algorithm attempts to overlap the execution of modules on P.host and on P.mem. To this end, the program is divided into two classes of regions. In a *module-wise parallel* region, there are multiple modules that can be run in parallel with respect to one another, while in a *module-wise serial* region, there is only one module that can be run at a time because of dependences between modules. The algorithms used in each type of region are different. In the following, we explain them. Note that, in these algorithms, we use basic modules. The reason is that basic modules are simpler and, therefore, expose more parallelism in the application. In addition, they are easier to parallelize because they are always loops.

Module-Wise Parallel Region.

The goal is to run some of the modules in this region on P.host and some on P.mem concurrently. We attempt to perform an initial partition of the modules as balanced as possible. We assign the modules to P.host or to P.mem based on their estimated affinity. However, if the resulting partition is not balanced according to static estimates (Section 3.2), we move some modules from the busier processor to the other one. If necessary, we also take the largest module remaining in the busier processor and partition it according to the algorithm for module-wise serial regions (See below).

If, at run-time, the load is imbalanced, we dynamically change the scheduling of some modules using algorithms similar to those in Section 3.3. If necessary, we also take the module that was partitioned and we dynamically repartition it according to the algorithm for module-wise serial regions.

Module-Wise Serial Region.

The goal is to partition the only module in this region between P.host and P.mem so that the load is as balanced as possible. In the following, we explain the general approach. The partition can be performed statically or dynamically, as explained in Section 3.5.1.

1. If the loop is fully parallel, we divide the iteration count into two chunks (Row 1 of Table 3). The sizes of the chunks are those that balance the load between P.host and P.mem, based on static or dynamic information.
2. Otherwise, if the loop is distributable across processors without synchronization, we distribute the loop (Row 2 of Table 3). Loop distribution splits the loop into the maximal strongly-connected components (called π-blocks) in the data dependence graph of the loop body [25]. Each component becomes a new loop. We then topologically sort the data dependence graph of the distributed loops and assign the loops to P.host or P.mem according to their estimated or real affinity. As usual, we try to balance the load based on static or dynamic information. When a final schedule is produced, all the loops assigned to a given processor are combined into a single one to reduce overhead.

   Note that, if the loop is both parallel and distributable, it is better to run it as a parallel loop than to distribute it. The reason is that parallelization tends to provide better cache reuse and makes it easier to balance the load.
3. Otherwise, if the loop can be distributed using dopipe scheduling [20], we do so (Row 3 of Table 3). The procedure is similar to the previous case. However, we now need to add synchronization between the processors, and write-back and invalidate commands to control P.host's cache. In the example in Row 3 of Table 3, P.host executes at least 4 iterations ahead of P.mem and uses *signal* and *wait* to synchronize. To keep the data coherent, before every synchronization, P.host writes back the updated cache lines.
4. Otherwise, we apply the best sequential scheduling strategy described in Section 3.3 to the loop. While we could still partition the loop and apply doacross scheduling in some cases, the synchronization overhead is likely to be too high [25].

Note that, when the compiler partitions a loop in cases 1 and 2, it must do so in such a way that P.host and P.mem do not falsely share any memory line. Otherwise, since the compiler is not inserting write-back or invalidation commands, there may be data coherence

Case	Original Loop	Partitioned Loop	
		P.host Code	P.mem Code
Fully Parallel	`DO I = 1, 100` `  B(I) = A(I)`	`DO I = 1, 70` `  B(I) = A(I)`	`DO I = 71, 100` `  B(I) = A(I)`

Distributable Without Synchronization	`DO I = 1, 100`
	`  A(I) = A(I-1)`
	`  C(I) = C(I+1)`
		`DO I = 1, 100`
		`  A(I) = A(I-1)`
			`DO I = 1, 100`
			`  C(I) = C(I+1)`

Distributable With Dopipe	`DO I = 1, 100`
	`  A(I) = A(I-1)+B(I)`
	`  C(I) = A(I)`
		`DO I = 1, 100`
		`  A(I) = A(I-1)+B(I)`
		`  IF (MOD(I,4).EQ.0) THEN`
		`  WRITEBACK(A(I-3) to A(I))`
		`  SIGNAL`
		`  ENDIF`
			`DO I = 1, 100`
			`  IF (MOD(I+3,4).EQ.0) THEN`
			`  WAIT`
			`  ENDIF`
			`  C(I) = A(I)`

Table 3: Partitioning a loop in a module-wise serial region.

problems. Therefore, the compiler must be aware of the data access patterns and data layouts when it partitions the loops.

3.5.1 Static and Dynamic Partitioning

All the cases in the algorithm for module-wise serial regions can use static or dynamic information to decide how to partition the module. The procedure is straightforward except that, because we may now be assigning small chunks of work, we need to be aware of cache write-back and invalidation overheads.

As an example, consider Case 1. To make the decision statically, we use the predicted execution time of the module on P.host (T_{phost}) and on P.mem (T_{pmem}), and the predicted number of iterations N. The obvious approach is to assign the iterations so that the load in P.host and in P.mem is balanced. This occurs when we assign N_{phost} iterations to P.host and N_{pmem} to P.mem such that:

$$\begin{aligned} N_{phost} &= \frac{N \times T_{pmem}}{T_{phost}+T_{pmem}} \\ N_{pmem} &= N - N_{phost} \end{aligned} \tag{3}$$

In this case, the estimated execution time of both P.host and P.mem will be $T_{total} = \frac{T_{phost} \times T_{pmem}}{T_{phost}+T_{pmem}} + T_{wbinv}$. In this formula, $T_{wbinv}(N_{pmem})$ is all the overhead involved in performing write-back and invalidation actions on P.host's cache. For simplicity, we assume that this overhead delays execution of both P.host and P.mem equally. T_{wbinv} can be estimated as a function of N_{pmem}, the number of iterations assigned to P.mem. Overall, in our algorithm, we use this partition unless T_{total} is larger than T_{phost}, in which case we execute the whole module on P.host.

To make the decision dynamically, we proceed similarly. In the first invocation of the loop, we use a certain partition of iterations, for example N_{phost} and N_{pmem}. In this first invocation, the run-time system measures the overhead-free execution time of these iterations, which is τ_{phost} and τ_{pmem}, respectively. In addition, it also measures the $T_{wbinv}(N_{pmem})$ overhead. With these measurements, the run-time system estimates the average execution time of one iteration on P.host (t_{phost}) and on P.mem (t_{pmem}) as: $t_{phost} = \frac{\tau_{phost}}{N_{phost}}$ and $t_{pmem} = \frac{\tau_{pmem}}{N_{pmem}}$. Based on these values, the run-time system partitions the loop in its next invocation in the program as follows. If the loop has N_{next} iterations, the assignment is:

$$\begin{aligned} N_{phost\_next} &= \frac{N_{next} \times t_{pmem}}{t_{phost}+t_{pmem}} \\ N_{pmem\_next} &= N_{next} - N_{phost\_next} \end{aligned} \tag{4}$$

unless T_{total_next} is larger than $t_{phost} \times N_{next}$, where $T_{total\_next} = t_{phost} \times N_{phost\_next} + T_{wbinv}(N_{pmem\_next})$. In this case, the whole module is executed on P.host.

3.6 Compiler Directives

Our system includes source-code compiler directives that allow the programmer to guide the algorithm. For example, they allow the programmer to identify modules and specify where and how they should be run. These directives are useful when the programmer knows the application well. A sample of our directives is shown in Table 4.

4 EVALUATION ENVIRONMENT

Compiler

We have implemented the compiler algorithm described in Section 3 so that it can be applied in a fully-automated manner. For the numerical applications, the algorithm is embedded in the Polaris parallelizing compiler [3]. Polaris takes Fortran programs and includes many compilation passes that our algorithm can benefit from. Such passes perform data dependence analysis, interprocedural analysis, symbolic analysis, and other operations. For the non-numerical applications, we cannot use Polaris and, therefore, apply our algorithm by hand.

Polaris helps identify with high accuracy the cache lines that have to be written back or invalidated from P.host's caches when execution is transferred between P.host and P.mem (Section 2.1). For the non-numerical applications, since we do not have tools to perform detailed data dependence analysis, we often conservatively write back or invalidate more cache lines than necessary.

Our system attempts to produce efficient code. Any module that is to be run on P.mem is bundled into a subroutine, which simplifies maintaining data coherence for register values. Moreover, the P.host and P.mem versions of a module are optimized for the processor they will run on. Specifically, P.host versions are loop-unrolled so that more ILP can be extracted dynamically and loads can be overlapped. For P.mem versions, we use blocking and loop distribution to minimize the pollution of the small P.mem cache.

Finally, a special case occurs if the loop in a module contains gotos that exit the module. In this case, the compiler transforms these gotos when the module is made into a subroutine. Specifically, new targets for these gotos are generated immediately after the loop in the same subroutine, and the subroutine is given one extra argument that is set to different values at these target positions. After executing the subroutine, the argument is checked by the caller of the subroutine and control branches to the original goto target in the caller according to the returned value of the argument. Multiple entries into the loop in a module are transformed by the compiler in a similar way.

Static Prediction

We set the parameters used in the static performance model to match the processor and system architectures modeled. Some of the most important parameters include the number and type of functional units in the processors, instruction latencies, cache sizes and organizations, and cache miss latencies. As indicated in Section 3.2, the model is currently designed only for numerical applications. For non-numerical ones, we predict based on data from two profiling runs of the code, one on P.host and one on P.mem, using a different input set than for the production run.

As indicated in Section 3.2.2, the static performance model returns undefined affinity for a module when the inaccuracy windows of the estimated P.host and P.mem execution times overlap. Based

Directive	Description
`C$PIM begin_module`	Marks the beginning of a module
`C$PIM end_module`	Marks the end of a module
`C$PIM on` *processor*	Run the module on *processor*
`C$PIM basic`	Use basic partitioning to extract modules
`C$PIM advanced`	Use advanced partitioning to extract modules
`C$PIM advanced_retraction`	Use advanced partitioning with retraction to extract modules
`C$PIM static` *inaccuracy_window*	Use static prediction with an inaccuracy window of *inaccuracy_window* to estimate the affinity of the module
`C$PIM coarse_basic`	Use coarse basic dynamic scheduling
`C$PIM coarse_recent`	Use coarse most recent dynamic scheduling
`C$PIM fine_basic`	Use fine basic dynamic scheduling
`C$PIM fine_first`	Use fine first invocation dynamic scheduling
`C$PIM partition` *host_iter*, *mem_iter*	Partition the following loop giving *host_iter* iterations to P.host and *mem_iter* to P.mem
`C$PIM partition_dynamic`	Partition the following loop dynamically between P.host and P.mem

Table 4: Sample of compiler directives.

on our experiments, we use $\pm 15\%$ inaccuracy windows. In addition, when the module so small that the model becomes less accurate, the affinity is also undefined. This occurs for modules whose estimated execution time on P.host or P.mem is lower than 50,000 cycles per invocation.

Finally, for very small modules, not even the profiles performed on non-numerical applications are accurate. The reason is that these profiles do not include the overheads to keep the data in the caches coherent or to bundle up the code into modules. We consider that profiled modules whose estimated execution time on P.host or P.mem is less than 2,000 cycles per invocation also have undefined affinity.

Applications

We evaluate both numerical and non-numerical applications. The numerical applications are: Swim and Mgrid from SPECfp2000, Tomcatv from SPECfp95, LU from [21], and TFFT2 from NAS [2]. The non-numerical ones are: Bzip2 and Mcf from SPECint2000 and Go and M88ksim from SPECint95. LU performs LU matrix decomposition. Table 5 shows the problem sizes used for the applications.

Application	Data Size and Number of Iterations
Swim	513×513, 10 iterations
Tomcatv	513×513, 5 iterations
LU	512×512
TFFT2	2^{17} elements, 5 iterations
Mgrid	$64 \times 64 \times 64$ grid, 3 iterations
Bzip2	512KB file (train, *test*, and a postscript file), level 7 compression + decompression
Mcf	test, *test with randomized edge weights*, test with pruned edges
Go	train (playlevel=20), *train (playlevel=50)*, train (playlevel=100)
M88ksim	test, *ref (dcrand) executing 50,000 instructions*, ref (dcrand) executing 300,000 instructions

Table 5: Applications used. All numerical applications use double precision. For each non-numerical application, we use three different inputs. Of these, the one used for profiling is shown in italics.

Simulation Environment and Architecture

The code generated by our algorithm is compiled into MIPS executable and run on a MINT-based [23] execution-driven simulation environment [15]. The simulation environment models dynamic superscalar processors with register renaming, branch prediction, and non-blocking memory operations [15]. The architecture modeled is that of Section 2, with a bus connecting the processor and memory chips. The architecture is modeled cycle by cycle, including contention effects. Table 6 shows the parameters used for each component of the architecture.

The L2 cache size used is 1 Mbyte for numerical applications and 512 Kbytes for non-numerical ones. We selected a smaller cache for non-numerical applications because they execute small problem sizes, especially the SPECint95 applications. Table 7 shows the L2 local hit rates of both types of applications. With 512-Kbyte L2 caches, the average hit rates of non-numerical applications get closer to those of numerical ones, which are around 80%.

Our choice of P.mem's clock frequency is motivated by recent advances in Merged Logic DRAM process. They appear to enable the integration of logic that cycles as fast as in a logic-only chip, with DRAM memory that is only 10% less dense than in a DRAM-only chip [9, 10].

The table also includes the overheads involved in invalidating and writing back lines from P.host's L2 cache. We assume the following hardware support in the L2 cache controller. If we want to write back num_cache_lines lines, P.host suffers an overhead of $5 + 1 \times num\_cache\_lines$ cycles to program the controller. Then, the controller writes back the desired lines in the background without stalling P.host. The write backs must be completed before passing execution to P.mem. If, instead, we want to invalidate num_cache_lines lines, P.host suffers a total overhead of $5 + 1 \times num\_cache\_lines$ cycles. These cycles can potentially be overlapped with P.mem execution.

Finally, P.host and P.mem synchronize at module boundaries as described in Section 2. The overheads involved in these synchronizations are considered in our simulation.

5 EVALUATION

To evaluate our algorithm, we first examine the characteristics of the modules (Section 5.1) and then evaluate non-overlapped execution with basic partitioning (Section 5.2) and advanced partitioning (Section 5.3), overlapped execution (Section 5.4), and overall speedups (Section 5.5).

5.1 Characteristics of the Modules

Table 8 shows the characteristics of the basic modules. The table has a section for the numerical applications and one for the non-numerical ones. The first row in each section shows the total number of basic modules in each application and their combined execution time relative to the total execution time of the application. All times are taken running on P.host. The second and third rows in each section break down the modules into whether or not they are parallelizable. The fourth and fifth rows break down the modules into whether they have overall affinity for P.host or P.mem. To compute the affinity of a module, we time the execution of all the invocations of the module on P.host and P.mem, and choose the processor with the lowest average time. The sixth row shows the average number of invocations for each individual module in a program. Finally,

Module	Parameter	Value
P.host :: P.mem	Frequency	800 MHz :: 800 MHz
	Issue Width	Out-of-order 6-issue :: In-order 2-issue
	Functional Units	4 Int + 4 FP + 2 Ld/St units :: 2 Int + 2 FP + 1 Ld/St units
	Pending Ld/St	8/16 :: 4/4
	Branch Penalty	4 cycles :: 2 cycles
P.host Caches	L1-Data	Write-through, 32-KB, 2-way, 32-B line, 2-cycle hit
	L2-Data	Write-back, 1-MB (512-KB for SPECint), 4-way, 128-B line, 10-cycle hit
	Write-Back Overhead	$5 + 1 \times num\_cache\_lines$ cycles to program. Actual data write back occurs in background
	Invalidation Overhead	$5 + 1 \times num\_cache\_lines$ cycles total. They may be overlapped with P.mem's execution
P.mem Cache	L1-Data	Write-back, 16-KB, 2-way, 32-B line, 2-cycle hit
Memory & Bus	Memory Latency	If row buffer miss: 160 cycles from P.host & 21 cycles from P.mem
		If row buffer hit: 152 cycles from P.host & 13 cycles from P.mem
	Bus Type	Split transaction, 16-B wide
	DRAM Memory Size	64 MB

Table 6: Parameters of the simulated architecture. Cache and memory latencies correspond to contention-free round-trips from the processor.

Numerical Apps.	L2 Hit Rate (1MB)	Non-Numerical Apps.	L2 Hit Rate (1MB)	L2 Hit Rate (512KB)
Swim	80.1%	Bzip2	61.8%	46.6%
Tomcatv	86.4%	Mcf	93.3%	85.3%
LU	57.2%	Go	100.0%	100.0%
TFFT2	86.8%	M88ksim	99.9%	99.9%
Mgrid	86.8%			
Average	79.5%	Average	88.8%	83.0%

Table 7: Comparing the L2 local hit rates of numerical and non-numerical applications.

the last row shows the average module size measured in number of P.host cycles that it takes to execute one invocation.

If we focus on the numerical applications, we see that there are only a few modules per application (5-21) and that they account for 99% of the application time. Parallel modules dominate in Swim, TFFT2, and Mgrid, while they have little weight in LU. In general, modules tend to be large and be invoked a low number of times.

In non-numerical applications, the number of basic modules is considerably higher (15-75). However, the coverage is low (on average 63% of the application time), especially in Go, where only 26% of the execution time is covered by basic modules. Serial modules dominate in all the non-numerical applications. Most of the modules in the non-numerical applications except Bzip2 are small and, not coincidentally, are invoked many times. Therefore, the overhead involved in executing the basic modules on different processors is likely to be large compared to their execution time. As a result, we may need advanced partitioning for these applications.

The table also shows that different applications have a different distribution of module affinity. While Swim, Tomcatv, Mgrid, and Mcf have mostly P.mem affinity, LU, Bzip2, Go, and M88ksim have mostly P.host affinity. TFFT2 has a balanced distribution of affinity.

5.2 Non-Overlapped & Basic

In this section, we evaluate the non-overlapped execution of the applications with basic partitioning. Since numerical applications and non-numerical ones have different behavior, we discuss them separately.

5.2.1 Numerical Applications

Figure 3 compares the execution times of the numerical applications under several scenarios. For each application, the three leftmost bars correspond to running the application on P.host alone (P.host(alone)), on P.mem alone (P.mem(alone)), and on an ideal non-overlapped environment (Ideal), respectively.

P.host(alone) and P.mem(alone) are obtained by running the *uninstrumented original applications* on either processor. Ideal is obtained by executing each module on the processor where, on average across all invocations, it runs the fastest. The code that is not covered by the basic modules is run on P.host. No overlap between P.host and P.mem execution is allowed. In this ideal environment, we do not consider any data coherence or message bundling overheads. However, we also disregard any dynamic variation of affinities because we fix the scheduling of modules to processors. Moreover, we do not consider any possibly good cache interactions between modules. Overall, Ideal is a good approximation to the lower bound for non-overlapped execution.

The remaining, thicker bars, correspond to real scenarios where part of the code is executed on P.host and part on P.mem. For now, we only consider the bars that use non-overlapped execution and basic partitioning: static scheduling (Static), coarse basic dynamic scheduling (Coarse), coarse most recent dynamic scheduling (CoarseR), fine basic dynamic scheduling (Fine), and fine first invocation dynamic scheduling (FineF).

In each application, all bars are normalized to P.host(alone). All non-ideal bars are divided into: execution of instructions (Busy), stall time due to memory accesses (Memory), stall time due to pipeline hazards (Other), time waiting for the other processor (Idle), and overhead due to write-back and invalidation of P.host caches (WB&INV). The thicker bars show the breakdown for P.host on the left side and the breakdown for P.mem on the right side.

P.host(alone) and P.mem(alone) show that the relative emphasis on computing and memory activity varies across applications: Swim, Tomcatv, and Mgrid run faster on P.mem, while LU runs faster on P.host and TFFT2 runs equally fast on both processors. These results are consistent with the module affinity of the applications shown in Table 8. Overall, they show that neither P.host nor P.mem is the best place to run all and every application.

As expected, Ideal is faster than both P.host(alone) and P.mem(alone). Ideal is relatively better in applications with a mixed affinity like TFFT2.

Static schedules modules according to the static performance model; if the latter returns undefined affinity for a module, the module runs on P.host. From the figure, we see that Static is somewhat close to Ideal for most applications except for TFFT2. Overall, Static is attractive because of its simplicity.

Coarse and CoarseR tend to be the best choices. They are usually

Characteristic (% of P.host Time)	Swim	Tomcatv	LU	TFFT2	Mgrid	Average
Total Modules	16 (100.0%)	7 (96.5%)	5 (100.0%)	17 (99.3%)	21 (99.8%)	13.2 (99.1%)
Parallel Modules	16 (100.0%)	5 (56.2%)	3 (7.4%)	15 (93.9%)	18 (96.9%)	11.4 (70.9%)
Serial Modules	0 (0.0%)	2 (40.3%)	2 (92.6%)	2 (5.4%)	3 (2.9%)	1.8 (28.2%)
Modules with P.host Affinity	1 (6.5%)	3 (18.2%)	2 (92.6%)	8 (44.0%)	6 (23.9%)	4.0 (37.0%)
Modules with P.mem Affinity	15 (93.5%)	4 (78.3%)	3 (7.4%)	9 (55.3%)	15 (75.9%)	9.2 (62.1%)
Average Number of Invocations	5.7	5.0	409.4	1,688.5	105.7	442.9
Average Module Size (P.host cycles)	9,319 K	9,081 K	1,952 K	1,923 K	575 K	4,570 K

Characteristic (% of P.host Time)	Bzip2	Mcf	Go	M88ksim	-	Average
Total Modules	75 (85.4%)	27 (81.0%)	50 (26.1%)	15 (59.6%)	-	41.8 (63.0%)
Parallel Modules	24 (0.7%)	6 (1.0%)	4 (0.8%)	1 (2.7%)	-	8.8 (1.3%)
Serial Modules	51 (84.7%)	21 (80.0%)	46 (25.3%)	14 (56.9%)	-	33.0 (61.7%)
Modules with P.host Affinity	51 (56.4%)	11 (14.3%)	50 (26.1%)	12 (58.8%)	-	31.0 (38.9%)
Modules with P.mem Affinity	24 (29.0%)	16 (66.7%)	0 (0.0%)	3 (0.8%)	-	37.8 (38.9%)
Average Number of Invocations	73,499	261,041	128,256	265,303	-	182,177
Average Module Size (P.host cycles)	1,822 K	84 K	<1 K	<1 K	-	477 K

Table 8: Characteristics of the basic modules that are invoked at least once. For the non-numerical applications, we use the profiling input data. For Go and M88ksim, we neglect subroutines whose combined contribution to the execution time forms the 5% tail.

as fast or faster than Ideal. The reason is that they use real measurement of execution times to run the modules on the processors adaptively. Unfortunately, in the process of doing so, they are likely to run each module sub-optimally at least once. This is the reason for the slowdown in Swim.

If we compare Coarse and CoarseR, we see that, in many applications, they behave similarly. The reason is that the workload of each individual module tends to remain constant across invocations. As a result, CoarseR does not offer any advantage over Coarse. However, an important exception is LU. In LU, several modules vary their workload across invocations gradually, allowing CoarseR to adapt. The result is that CoarseR is about 20% faster than Coarse. While we recommend to use CoarseR, we note that it can sometimes be slower than Coarse. Specifically, abrupt changes in a module's workload across invocations may confuse CoarseR. Such behavior occurs in TFFT2.

The fine strategies are not as attractive. Fine is sometimes slow because of the high overhead resulting from its frequent decision runs (TFFT2). As for FineF, although it has the lowest decision run overhead of all the dynamic schemes, it often suffers because all decisions are made based exclusively on the first two iterations of the first invocation of the module. Consequently, unless the workload of the module is constant across invocations and iterations, the decision is likely to be sub-optimal (Tomcatv).

Overall, we conclude that, for numerical applications, CoarseR is the best strategy under non-overlapped execution and basic partitioning. CoarseR is very close to Ideal and, on average, it is 20% faster than P.host(alone) and 12% faster than P.mem(alone). We also conclude that the write-back and invalidation overheads are negligible.

5.2.2 Non-Numerical Applications

Figure 4 compares the execution times of the non-numerical applications under different scenarios and input sets. For each application, we consider two different input sets, namely input1 and input2. These inputs are different from the one used in the profiling run that provides data for the static predictions. In each application and input set, the bars are normalized and broken down into categories as in Figure 3.

As in the numerical applications, the P.host(alone) and P.mem(alone) bars show that neither P.host nor P.mem is the best place to run all and every application. Go, M88ksim and, to a lesser extent, Bzip2 run faster on P.host, while Mcf runs faster on P.mem. These results are expected from Table 8. The bars also confirm that Ideal is often much faster than P.host(alone) and P.mem(alone).

The main characteristic of non-numerical applications is that the Static, Coarse, CoarseR, Fine, and FineF strategies evaluated in Figure 3 for numerical applications perform very poorly now. Although not shown in the figure, these strategies take 11-100% longer than P.host(alone) to complete execution. This effect is caused by the small size of basic modules in non-numerical applications (Table 8). They are so small that overheads due to data coherence, code bundling, and instrumentation affect execution time significantly.

5.3 Non-Overlapped & Advanced

We now evaluate the non-overlapped execution of the applications with advanced partitioning. As usual, we consider numerical and non-numerical applications separately.

5.3.1 Numerical Applications

If we apply advanced partitioning to the numerical applications, the average number of modules per application decreases only slightly. Moreover, practically all the resulting compound modules are invoked more than twice, which makes retraction as discussed in Section 3.1.3 not applicable. With these compound modules, we re-execute the applications following the coarse basic and coarse most recent dynamic scheduling strategies. The result, shown in Figure 3, are bars AdvCoarse and AdvCoarseR, respectively. Recall that fine strategies are not applicable under advanced partitioning.

If we compare bars AdvCoarse and AdvCoarseR to Coarse and CoarseR in Figure 3, we see that advanced partitioning has little impact on the performance of numerical applications. In fact, it only manages to speed up TFFT2 by 5-7%. The reasons for the marginal improvement are that there is little reuse of data across the combined modules, and that overheads are an insignificant component of the execution time.

5.3.2 Non-Numerical Applications

If we apply advanced partitioning to the non-numerical applications, we reduce the number of modules per application significantly. Moreover, many of the resulting compound modules are invoked less than three times, which means that we can optionally apply retraction. Consequently, we take the compound modules after retraction and re-evaluate three strategies: static scheduling, coarse basic dynamic scheduling, and coarse most recent dynamic scheduling. The result, shown in Figure 4, are bars AdvStatic, AdvCoarse, and AdvCoarseR, respectively. We have also evaluated the strategies without retraction, which we discuss later.

The figure shows that AdvStatic can perform close to Ideal in some special cases. One case is when the production and profiling runs use similar input sets, as in input1 of Bzip2. Another case is when modules do not change affinities for different input sets. This occurs

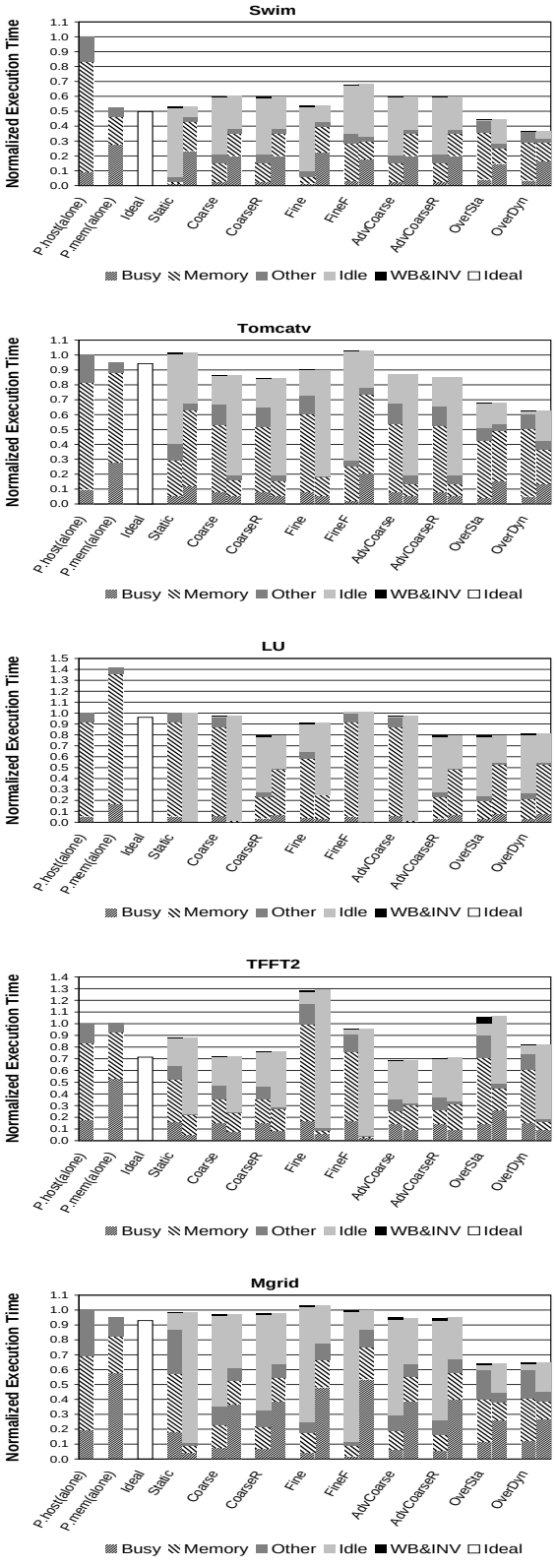

Figure 3: Execution time of the numerical applications under different scenarios.

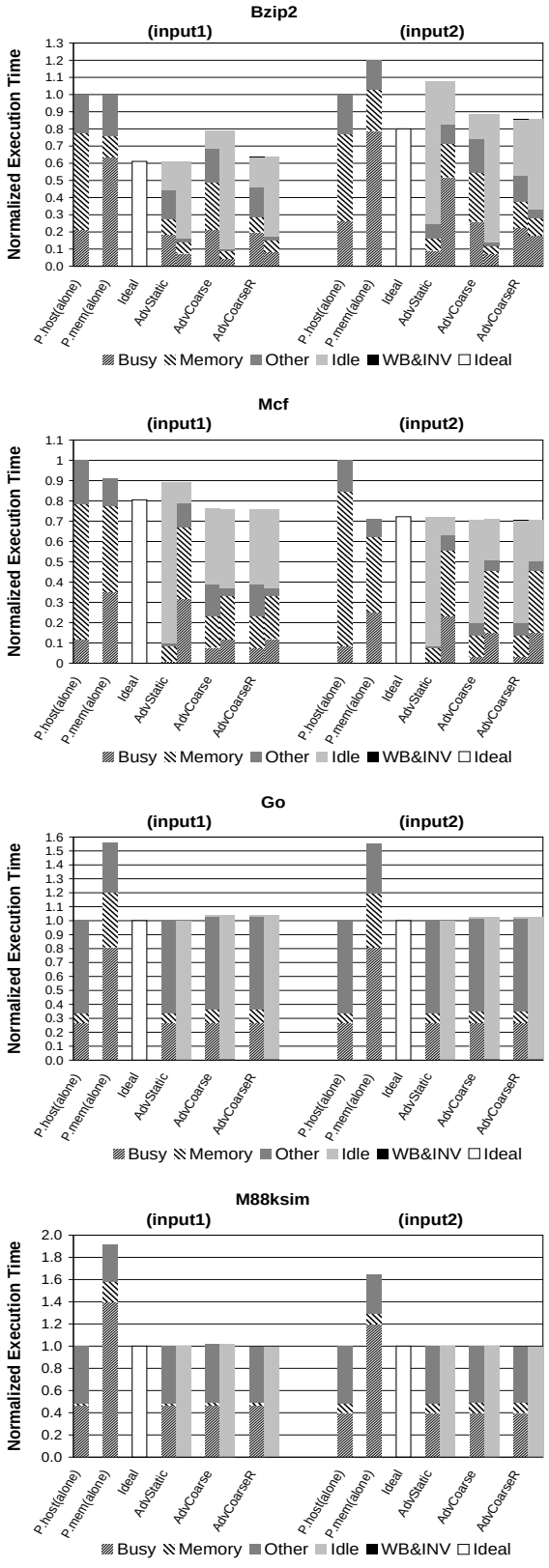

Figure 4: Execution time of the non-numerical applications under different scenarios.

in Go and M88ksim, where all modules have P.host affinity because the working set fits in the L2 cache for all the input sets evaluated. In general, however, AdvStatic is not a good choice because it can perform very poorly, as in input2 of Bzip2 and input1 of Mcf.

The adaptability of AdvCoarse and AdvCoarseR to run-time conditions enables them to perform well when the production and profiling runs are very different. This occurs in input2 of Bzip2 and input1 of Mcf. Moreover, AdvCoarseR is more robust than AdvCoarse. This can be seen in input1 of Bzip2, where the workload of modules changes across invocations. Consequently, AdvCoarseR is our best choice. On average, it is about as fast as Ideal, while it runs 12% and 30% faster than P.host(alone) and P.mem(alone), respectively.

Although not shown in Figure 4, not using retraction delivers worse results. Specifically, if we do not apply retraction, AdvCoarse and AdvCoarseR do not change much in most cases. However, they become as slow as AdvStatic in input2 of Bzip2. The reason is that they are not exploiting all the potential of adaptive strategies. Therefore, we use retraction.

5.4 Overlapped Execution

Finally, we overlap the execution of P.host and P.mem according to the algorithm in Section 3.5. We attempt the two approaches discussed in that section, namely static and dynamic scheduling. The result is shown in Figure 3 as bars OverSta and OverDyn, respectively. We do not show data for the non-numerical applications because our algorithm is currently unable to overlap P.host and P.mem execution in those applications.

Recall that overlapping is applied over basic partitioning only. In the numerical applications, the algorithm finds many module-wise serial regions but no significant module-wise parallel region. In the different module-wise serial regions, it attempts to partition all 66 basic modules. Of those, 57 modules are partitioned using iteration parallelism (Case 1 in the algorithm of Section 3.5) and 2 using loop distribution (Case 2). The remaining 7 are not partitioned.

Figure 3 shows that, for the majority of applications, overlapped execution is significantly faster than the best non-overlapped strategy, either real (AdvCoarseR) or not (Ideal). Specifically, in Swim, Tomcatv, and Mgrid, the overlapped strategies OverSta and OverDyn are 30-40% faster than AdvCoarseR. With overlapped strategies, we are utilizing processor resources that would otherwise remain idle.

The behavior of the other applications is easy to explain. In LU, the strategies for overlapped execution have no effect because the most significant modules have dependences and cannot be partitioned. TFFT2, on the other hand, runs slower because the strategies induce extra overhead. While some of the overhead comes from the additional code bundling and instrumentation added, most of it is induced by the need to ensure coherence for the cached data. For example, P.host needs to execute instructions to write back and invalidate cached data structures when execution is transferred to P.mem and back (WB&INV in Figure 3). P.host also executes additional instructions to generate the addresses of these data structures. These instructions add to P.host's Busy in Figure 3, which does not appear to have increased because, at the same time, much work has been transferred to P.mem. Finally, extra misses in the caches of P.host occur when, after the cache invalidations, P.host reloads the data (higher Memory Stall in Figure 3).

Fortunately, the same chart for TFFT2 shows that, while OverSta suffers greatly from these overheads, OverDyn is able to eliminate most of them. As a result, OverDyn only takes 12% longer than AdvCoarseR to execute TFFT2. OverDyn is better because it is aware of the extra overheads induced by overlapped execution and schedules modules more conservatively. Overall, taking the average over all numerical applications, OverDyn is over 15% faster than AdvCoarseR, the best strategy for non-overlapped execution.

5.5 Overall Speedups

The results from the previous sections indicate that the best overall strategy for numerical applications is the OverDyn overlapped execution. However, overlapped execution does not improve over non-overlapped execution for our non-numerical applications. The best non-overlapped execution strategy in both types of applications is AdvCoarseR, which is applied with retraction when applicable.

To summarize our results, Table 9 compares the speedups for different environments. Relative to P.host(alone), AdvCoarseR delivers an average speedup of 1.31 and 1.18 for numerical and non-numerical applications, respectively. Relative to P.host(alone), OverDyn delivers an average speedup of 1.66 for numerical applications. Recall that P.host(alone) uses the original, uninstrumented code.

In general, we expect individual numerical applications to benefit from overlapped execution, unless they are highly serial (LU) or suffer from the data coherence and related overheads of overlapped execution (TFFT2). Furthermore, we expect individual non-numerical applications to benefit from non-overlapped execution strategies like AdvCoarseR to a larger degree than our average results indicate. Our average is pulled down by the fact that Go and M88ksim have working sets that fit in P.host's caches and, therefore, all their modules have P.host affinity. In this case, our strategies cannot help.

To put our results in perspective, Table 9 shows the speedups under two related scenarios. The first one is an *ideal* system with 2 P.hosts. To obtain it, we use the Polaris [3] and the Silicon Graphics parallelizing compilers to identify the parallel sections in the numerical and non-numerical applications, respectively. Then, we compute the *IdealAmdahl* execution time of the applications by taking P.host(alone) and dividing by two the time taken by the code marked parallel.

We can see from the table that our algorithm delivers speedups that are comparable to this *ideal* speedup. Indeed, with our algorithm, the speedups of the numerical applications are nearly as high, while the speedups of the non-numerical ones are higher. We note that the 2-P.host system, in addition to being ideal, can be argued to be more expensive than our heterogeneous system.

The second related scenario is a real 2-processor Silicon Graphics Challenge. We ran the numerical and non-numerical applications on such a machine after the Polaris and the Silicon Graphics parallelizing compilers, respectively, had marked the parallel sections. The speedups on such a machine are shown in Table 9. Clearly, the results are not directly comparable to our simulation results because of differences in architecture. However, they give an idea of how parallelizable these applications are.

6 RELATED WORK

Many different types of processor in memory or intelligent memory architectures have been proposed, including IRAM [14], Shamrock [13], Raw [24], Smart Memories [16], Imagine [22], FlexRAM [12], Active Pages [19], and DIVA [8] among others. Some of these architecture projects have associated compiler efforts [1, 5, 8, 11, 18].

In this paper, we focus on an architecture with host and memory processing. Past work for these architectures [5, 8, 12, 19] largely assumes that the programmer isolates the code sections to run on the memory processors. In addition, that work has mostly focused on executing sections of code on only a set of identical memory processors. Our work, instead, tries to automatically partition the code into sections and then schedule each section on its most suitable processor. Another characteristic of our approach is that it uses a combination of compiler and run-time algorithms to map the code.

Exploiting parallel execution in a heterogeneous environment has been proposed in systems like Globus [6] or Legion [7] among others. Such work is quite different from ours. It uses bigger module grain sizes and targets highly-distributed systems.

Numerical Applications	$\frac{P.host(alone)}{AdvCoarseR}$	$\frac{P.host(alone)}{OverDyn}$	$\frac{P.host(alone)}{IdealAmdahl}$	2-Proc. SGI	Non-Numerical Applications	$\frac{P.host(alone)}{AdvCoarseR}$	$\frac{P.host(alone)}{IdealAmdahl}$	2-Proc. SGI
Swim	1.67	2.71	2.00	1.85	Bzip2	1.37	1.01	0.99
Tomcatv	1.17	1.60	1.67	1.44	Mcf	1.37	1.01	1.00
LU	1.26	1.22	1.04	0.99	Go	0.97	1.01	0.57
TFFT2	1.42	1.22	1.91	0.80	M88ksim	1.01	1.03	1.00
Mgrid	1.05	1.55	1.94	1.47				
Average	1.31	1.66	1.71	1.31	Average	1.18	1.02	0.89

Table 9: Comparison of speedups for different environments. The speedups of the non-numerical applications correspond to the average of the runs for the two input sets that are not used for profiling.

7 CONCLUSIONS AND FUTURE WORK

This paper has presented and evaluated an algorithm to automatically map code to a heterogeneous architecture composed of a host processor and a simpler memory processor. By carefully partitioning the code and scheduling it on the most suitable processor, we demonstrated speedups that are as high and often higher than *ideal* speedups on a more expensive 2-host multiprocessor system.

Overall, our results indicate that a heterogeneous mix of processors is a promising approach to speed-up applications cost-effectively. The cost-effectiveness of architectures similar to the one considered here is likely to be higher than that of conventional multiprocessors with homogeneous processors.

We see this work as only a first step toward effectively mapping code on this class of intelligent memory architectures. Proposed systems like Active Pages [19], DIVA [8], and FlexRAM [12] all have several processors per memory chip and several memory chips. Consequently, we are in the process of extending our software infrastructure with code parallelization algorithms and data mapping techniques to support such architectures.

ACKNOWLEDGMENTS

We thank Paul Feautrier, Jose Renau, and David Padua for contributions to this work.

REFERENCES

[1] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 4–15, May 1999.

[2] NAS Parallel Benchmark. http://www.nas.nasa.gov/pubs/techreports/nasreports/nas-98-009/.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L.Rauchwerger, and P.Tu. Parallel Programming with Polaris. *IEEE Computer 29(2)*, pages 78–83, December 1996.

[4] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Twelfth International Workshop on Languages and Compilers for Parallel Computing*, 1999.

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

[6] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

[7] A. S. Grimshaw, W. A. Wulf, and the Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[8] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. In *Supercomputing 1999*, November 1999.

[9] IBM Microelectronics. Blue Logic SA-27E ASIC. In *News and Ideas of IBM Microelectronics*, page 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*, April 1999.

[11] D. Judd and K. Yelick. Exploiting On-Chip Memory Bandwidth in the VIRAM Compiler. In *2nd Workshop on Intelligent Memory Systems*, November 2000.

[12] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the International Conference on Computer Design*, October 1999.

[13] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

[14] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, September 1997.

[15] 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 (PACT)*, October 1998.

[16] K. Mai, T. Paaske, N. Jayasena, R. Ho, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[17] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.

[18] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. Technical Report LCS-TM-599, Massachusetts Institute of Technology, August 1999.

[19] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, June 1998.

[20] D. A. Padua. Multiprocessors: Discussion of Some Theoretical and Practical Problems. Technical Report UIUCDCS-R-79-990, Department of Computer Science, The University of Illinois at Urbana-Champaign, November 1979.

[21] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran 77*. Cambridge University Press, 1992.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *31st International Symposium on Microarchitecture*, November 1998.

[23] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[24] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it All to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.

[25] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesley, 1991.