Automatic Code Mapping on an Intelligent Memory Architecture

Yan Solihin, *Student Member*, *IEEE*, Jaejin Lee, *Member*, *IEEE*, and Josep Torrellas, *Senior Member*, *IEEE*

Abstract—This paper presents an algorithm to automatically map code on a generic intelligent memory system that consists of a high-end host processor and a simpler memory processor. To achieve high performance with this type of architecture, the 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 obtain average speedups of 1.7 for numerical applications and 1.2 for nonnumerical 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 heterogeneous intelligent memory systems.

Index Terms—Intelligent memory architecture, processing-in-memory, compilers, adaptive execution, performance prediction, heterogeneous system.

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 processing-in-memory, has been recently proposed in several research systems [8], [12], [13], [14], [15], [18], [23], [26], [28] and used in a few commercial parts [20], [22].

Some proposals use this architecture for the main processing unit in the system. Examples of such systems are EXECUBE [13] and successors [14], IRAM [15], Raw [28], and Smart Memories [18], among others. Other proposals instead use this architecture for the memory system to replace plain memory chips. In this case, intelligent memory chips act as coprocessors in memory that execute code when signaled by the host (main) processor. Examples of proposed systems that follow this approach are Active Pages [23], FlexRAM [12], and DIVA [8].

In this second class of systems, we have a heterogeneous mix of processors: host and memory processors. A host processor is a state-of-the-art high-end processor. It is backed up by a deep cache hierarchy and suffers a high latency to access memory. A memory processor is typically less powerful. However, it 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 map code to these systems?

In previous work on these systems [5], [8], [12], [23], 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. Such an approach is often not much different from running code on a conventional parallel processor.

Our goal, instead, is to automatically partition the code into homogeneous sections and then schedule each section on the most suitable processor, while encouraging 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 partitions, maps, and schedules code on 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 obtain average speedups of 1.7 for numerical applications and 1.2 for nonnumerical 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

- *Y. Solihin and J. Torrellas are with the Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801. E-mail: {solihin, torrellas}@cs.uiuc.edu.*
- *J. Lee is with the Department of Computer Science and Engineering, Michigan State University, 2138 Engineering Bldg., East Lansing, MI 48823. E-mail: jlee@cse.msu.edu.*

Manuscript received 5 Dec. 2000; revised 25 May 2001; accepted 31 May 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114260.

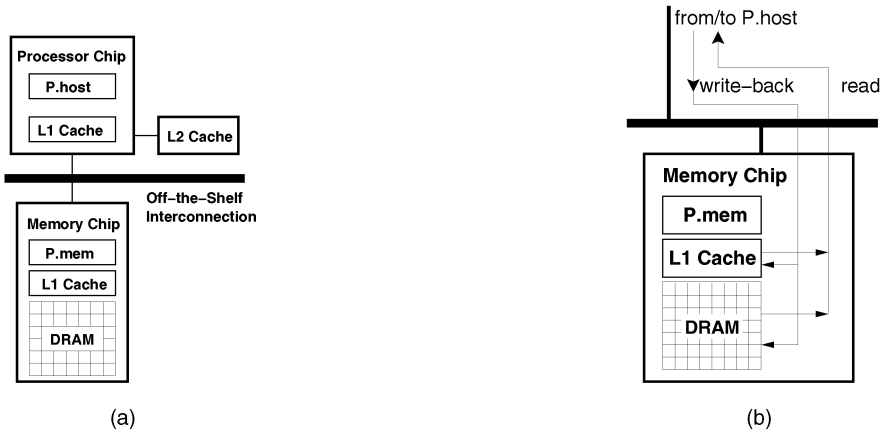

Fig. 1. Intelligent memory architecture considered.

toward effectively mapping code on heterogeneous 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 paper, we use a simpler architecture with only one processor of each type (Fig. 1a). Supporting multiple processors of each type requires extending the techniques that we will present, possibly by augmenting them with conventional parallelization techniques.

P.host is a wide-issue, high-end superscalar processor with a deep cache hierarchy and a high memory access latency. P.mem, instead, is a simple, narrow-issue superscalar processor 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 be 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. Such hardware would likely increase the cost of the system substantially and require a custom-designed interconnect.

We assume some simple support in P.mem's cache that makes programming easier (Fig. 1b). 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 data as it is being transferred if it has a copy of the line.

For P.host and P.mem to execute an application together, 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 off-load 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 own 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 need to 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 rereferences lines written by P.mem, it will miss in its caches, giving an opportunity to P.mem's cache to overwrite the returning data.

Therefore, in the general case, transferring execution from P.host to P.mem and then back to P.host induces three types of overhead 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 depends heavily 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 also 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 also schedule the cache 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. In all cases, if the compiler cannot accurately identify the lines to write back or invalidate, it has to conservatively write back or invalidate a superset of them.

3 An Algorithm to Map the Code

We have implemented a compiler and runtime algorithm that automatically maps applications to the intelligent memory architecture of Section 2. The algorithm maps both numerical and nonnumerical 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, the compiler estimates which processor each module should run on (Section 3.2). Since static scheduling decisions may be poor, the algorithm can also insert code that identifies at runtime 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 scheduling units, which we call *modules*. For best performance, a module should have a homogeneous (i.e., not time-varying) computing and memory behavior within an invocation. Ideally, it should also have good locality and be easy to extract from the code. We use two partitioning algorithms of increasing complexity: *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. By allowing only one loop per level, we increase the chances of finding a homogeneous behavior. Note that 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 in program order. 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 after M, 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 the 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 Fig. 2a 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 Fig. 2b.

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 overhead 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 the overhead, 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 in program order. The algorithm then works on one subroutine at a time. For a given subroutine, it applies an expansion subalgorithm to every module in sequence. Then, it applies a combining subalgorithm to every module in sequence. The two subalgorithms are repeatedly applied until the modules in the subroutine do not change further.

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

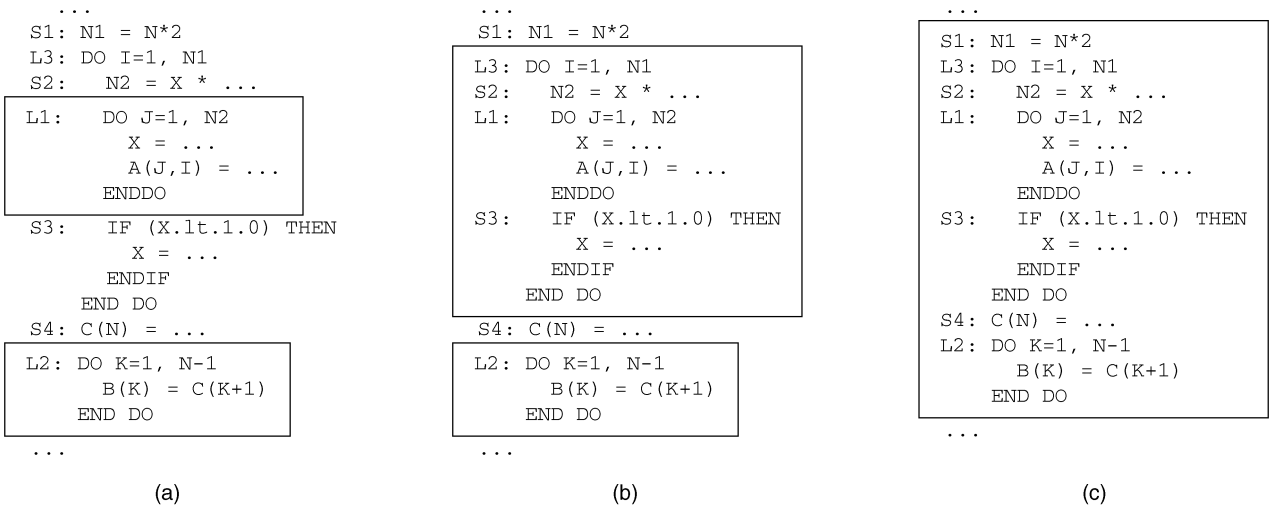

Fig. 2. A sample code with two loops marked as initial modules (a) and its structure after basic partitioning (b) and advanced partitioning (c). We assume that the two modules in (b) have the same estimated affinity.

The expansion subalgorithm 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 subalgorithm 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 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, Fig. 2c shows the result of applying advanced partitioning to the code of Fig. 2b. If we assume that the two basic modules have the same estimated affinity, the whole code becomes a compound module.

Note that, for small modules, the overhead that they will induce may overwhelm their execution time. We will see in Section 3.2.2 that these modules are statically assigned *undefined* affinity. In the advanced partitioning algorithm, if 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, they get combined. If two small modules are considered for combination, they are also combined. However, 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. In this case, runtime adaptation (Section 3.3) may not be applicable 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 we reach an all-enclosing loop or a set of disjoint loops. In this 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 runtime 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 an 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 applications, which are usually easier to analyze. For nonnumerical 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 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, namely computing time (T_{comp}) and memory stall time ($T_{memstall}$), and adding them up.

To estimate T_{comp}, we proceed as follows: Each instruction in the module is multiplied by its execution latency. The results for all instructions are grouped into three execution times: time of integer instruction execution (T_{int}), floating-point instruction execution (T_{fp}), and load/store instruction 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 during 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:

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

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 a module is estimated as follows: The data dependence structure of the code is analyzed to predict the access patterns. The latter then drive a stack-distance model of the cache [4], [19]. 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 surely overlapped with computation. In addition, in a real execution, resource contention increases the stall. For simplicity, in our calculations, we use as stall time per miss the full cache miss penalty, therefore assuming no overlap and no contention. In practice, the approach described seems to estimate the total memory stall time reasonably accurately. This outcome may be the result of overestimating the stall time per miss and underestimating the number of misses.

3.2.2 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 is unavailable at compile time, like the outcomes of branches. 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.

TABLE 1
Estimating the Affinity of a Module

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

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 estimated value. Therefore, we have an inaccuracy window of $\pm err\%$. As a result, 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). In practice, with the inaccuracy window that we use (Section 4), the estimated affinity of a module is usually the same as the real affinity.

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

3.3 Adaptive Execution

Our 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, the compiler inserts code in the module to measure, at runtime, the execution time of some of its invocations or (if applicable) some of its loop iterations. These invocations or iterations where execution times are measured are called *decision runs*. Based on the measurements in the decision runs, the module determines its affinity and schedules its remaining invocations or loop iterations 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 in the module. 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

TABLE 2
Comparing the Different Adaptive Scheduling Strategies

Num. Module 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	—

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 schedule for the remaining invocations of the module in the program is not fixed at that point. Instead, every time 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 at every invocation of the module. 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 eight 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 scheduling strategy.

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 competitive for modules invoked very few times.

Fine strategies, on the other hand, are 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, for example, in 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 that remain with *undefined* affinity after advanced partitioning are always run on P.host. This approach is likely to reduce the overhead associated with transferring execution between processors. For a module with defined estimated 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. In doing so, we could use many conventional compiler techniques for parallelization. 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 of them 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

TABLE 3
Partitioning a Loop in a Module-Wise Serial Region

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)`

into two classes of regions, namely *module-wise parallel* and *module-wise serial* regions.

In a module-wise parallel region, there are multiple modules that can be run in parallel with respect to one another. 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 discuss 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, since they are loops, they are easier to parallelize.

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 start by statically assigning the modules in the region to P.host or to P.mem based on their estimated affinity. The resulting partition is likely to be fairly imbalanced according to static estimates (Section 3.2). Consequently, we balance the load by statically moving modules between processors. In addition, we partition the largest module remaining in the busier processor. The partitioning can be done either statically or dynamically according to the algorithm for module-wise serial regions described below.

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 partitioning approach. Note that the partition can be performed statically or dynamically, as we discuss 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 [29]. 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 fine-tune load balancing.
3. Otherwise, if the loop can be distributed using dopipe scheduling [24] and the resulting partitions are not highly imbalanced, 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 invalidation commands to control P.host's caches. In the example in Row 3 of Table 3, P.host executes at least four iterations ahead of P.mem. The processors use *signal* and *wait* to synchronize. To keep the data coherent, P.host writes back the updated cache lines before every synchronization.
4. Otherwise, we schedule the loop using the best nonoverlapped strategy described in Section 3.3. While we could still attempt to partition the loop and use doacross scheduling in some cases, the synchronization overhead is likely to be too high [29].

Recall that the machine does not support hardware cache coherence. Consequently, 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 in the loop, there may be data coherence problems. Therefore, the compiler must be aware of the data access patterns and data layouts when it partitions the loops in Cases 1 and 2.

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 be assigning small chunks of work, we need to be aware of the cache write-back and invalidation overhead.

TABLE 4
Sample of Compiler Directives

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$PIM static` *inaccuracy_window*	Use the static performance model with an inaccuracy window of *inaccuracy_window* to estimate the affinity of the module
`C$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

Use of these directives is optional. They have not been used in the experiments described in this paper.

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. We want 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:

$$N_{phost} = \frac{N \times T_{pmem}}{T_{phost} + T_{pmem}}, \quad N_{pmem} = N - N_{phost}. \qquad (3)$$

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, $T_{wbinv} = f(N_{pmem})$ is all the overhead involved in performing write-back and invalidation actions on P.host's cache. These write-back and invalidation actions are performed before and while P.mem executes, respectively. 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. If, due to the T_{wbinv} overhead, T_{total} is larger than T_{phost}, we execute the whole module on P.host.

To make the decision dynamically, we proceed as follows: In the first invocation of the loop, we use the partition decided statically as indicated above (N_{phost} and N_{pmem}). In this first invocation, the module measures the overhead-free execution time of these iterations, which we call τ_{phost} and τ_{pmem}, respectively. In addition, it also measures the $T_{wbinv} = f(N_{pmem})$ overhead. With these measurements, the module 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 runtime system knows how to partition the loop in its next invocation in the program. If the loop has N_{next} iterations, the new assignment is:

$$N_{phost\_next} = \frac{N_{next} \times t_{pmem}}{t_{phost} + t_{pmem}}, \quad N_{pmem\_next} = N_{next} - N_{phost\_next} \qquad (4)$$

and $T_{total\_next} = t_{phost} \times N_{phost\_next} + T_{wbinv}$. As usual, if the $T_{wbinv} = f(N_{pmem\_next})$ overhead is expected to make T_{total_next} larger than $t_{phost} \times N_{next}$, we instead execute the whole module on P.host.

3.6 Compiler Directives

Our system includes several optional 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. Use of these directives is optional. They have not been used in the experiments described in this paper.

4 Evaluation Environment

4.1 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 contains several compilation passes that our algorithm can benefit from. Such passes perform data dependence analysis, interprocedural analysis, symbolic analysis, and other operations. For the nonnumerical applications, we cannot use Polaris because they are written in C. Consequently, we 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). If such cache lines cannot be identified accurately, they are selected conservatively by Polaris. For the nonnumerical 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

TABLE 5
Applications Used

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	512 KB 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

All numerical applications use double precision. For each nonnumerical application, we use three different inputs. Of these, the one used for profiling is shown in italics.

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.

4.2 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 nonnumerical 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 on our experiments, we use ± 15 percent inaccuracy windows. In addition, when the module is 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 nonnumerical applications are accurate. The reason is that these profiles do not include the overhead 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.

4.3 Applications

We evaluate both numerical and nonnumerical applications. The numerical applications are: Swim and Mgrid from SPECfp2000, Tomcatv from SPECfp95, LU from [25], and TFFT2 from NAS [2]. The nonnumerical 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.

We selected applications with a variety of behaviors. Among the numerical applications, some are fairly parallel (Swim, Tomcatv, TFFT2, and Mgrid), while others are largely serial (LU). Among the nonnumerical ones, some are memory bound (Bzip2 and Mcf), while others have working sets that largely fit in caches (Go and M88ksim).

4.4 Simulation Environment and Architecture

The code generated by our algorithm is compiled into MIPS executable and run on a MINT-based [27] execution-driven simulation environment [16]. The simulation environment models dynamic superscalar processors with register renaming, branch prediction, and nonblocking memory operations [16]. 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 nonnumerical ones. We selected a smaller cache for nonnumerical 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 local hit rate of nonnumerical applications gets closer to that of numerical ones, which is around 80 percent.

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 percent less dense than in a DRAM-only chip [9], [10].

The table also includes the overhead involved in invalidating and writing back lines from P.host's L2 cache. We assume the following hardware support in the on-chip L2 cache

TABLE 6
Parameters of the Simulated Architecture

Module	Parameter	Value
Processor	Frequency	P.host: 800 MHz, Pmem: 800 MHz
	Issue Width	P.host: out-of-order 6-issue, P.mem: in-order 2-issue
	Functional Units	P.host: 4 Int + 4 FP + 2 Ld,St units, P.mem: 2 Int + 2 FP + 1 Ld,St units
	Pending Ld,St	P.host: 8,16, P.mem: 4,4
	Branch Penalty	P.host: 4 cycles, P.mem: 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. 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, 200 MHz

Cache and memory latencies correspond to contention-free round-trips from the processor. The cycles listed for write-back and invalidation overhead are those seen by the processor.

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. 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 overhead involved in this synchronization is considered in our simulations.

5 Evaluation

To evaluate our algorithm, we first examine the characteristics of basic modules (Section 5.1). We then evaluate nonoverlapped 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 nonnumerical 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 for 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 the application. Finally, 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 an average of 99.1 percent 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 to be invoked only a few times. On average, a module takes 4.57 M cycles to execute and is invoked 443 times.

In nonnumerical applications, the number of basic modules is considerably higher (15-75). However, they account for only a modest fraction of the application execution time (on average, 63.0 percent). In Go, only 26.1 percent of the execution time is covered by basic modules. Serial modules dominate in all the nonnumerical applications. Most of the modules in the nonnumerical applications, except Bzip2, are small and, not coincidentally, are invoked many times. On average, a module takes 0.477 M cycles to execute and is invoked 182 K times.

TABLE 7
Comparing the L2 *local* Hit Rates of Numerical and Nonnumerical Applications

Numerical Apps.	L2 Hit Rate (1 MB)	Non-Numerical Apps.	L2 Hit Rate (1 MB)	L2 Hit Rate (512 KB)
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 8
Characteristics of the Basic Modules

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%)	-	10.8 (24.1%)
Average Number of Invocations	73,499	261,041	128,256	265,303	-	182,025
Average Module Size (P.host cycles)	1,822 K	84 K	<1 K	<1 K	-	477 K

For the nonnumerical applications, we use the profiling input data. For Go and M88ksim, we neglect subroutines whose combined contribution to the execution time forms the 5 percent tail.

Therefore, the overhead involved in executing basic modules 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. Swim, Tomcatv, Mgrid, and Mcf have mostly P.mem affinity. LU, Bzip2, Go, and M88ksim have mostly P.host affinity. Finally, TFFT2 has a balanced distribution of affinity.

5.1.1 Static Performance Model Results

Another interesting aspect of basic modules is whether or not their affinity can be predicted by our static performance model. Table 9 classifies the basic modules of numerical applications into those which the model can be applied to (*Can Be Applied*) and the others (*Cannot Be Applied*). When the model can be applied, the result may be insufficient accuracy to issue an affinity prediction, a correct affinity prediction, or an incorrect affinity prediction. Insufficient accuracy occurs when the inaccuracy windows of P.host and P.mem overlap. Finally, when the model cannot be applied, it may be because the module is too small, in which case, the affinity is undefined, or because there is insufficient dependence information at compile time. For each category, the table shows the number of modules and their weight in percentage of the execution time of the application on P.host.

The table shows that the modules to which the model can be applied usually have the highest weight (69.2 percent on average). LU is an exception: The model cannot be applied to any module because there is insufficient dependence information available at compile-time. The estimated affinity is correct in modules that account for an average of 29.2 percent of the application time. Since we use a conservative inaccuracy window, a few large modules have undefined affinity (1.4 modules with a 35.0 percent weight on average). Finally, incorrect affinity estimation is rare since, on average, only 0.8 modules with a 5.0 percent weight fall into this case.

The modules which the model cannot be applied to have relatively less weight (29.9 percent on average). Small modules are not a problem since they cover on average less than 3 percent of P.host time. However, insufficient compile-time information disables application of the model to all modules in LU and one large module in TFFT2.

5.2 Nonoverlapped and Basic

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

TABLE 9
Results of Applying the Static Performance Model to Basic Modules

Module Type	Number (% of P.host Time)	Swim	Tomcatv	LU	TFFT2	Mgrid	Average
Model Can Be Applied	Total	6 (97.3%)	5 (96.2%)	0 (0.0%)	13 (64.5%)	8 (88.2%)	6.4 (69.2%)
	Insufficient Accuracy	1 (29.1%)	1 (49.4%)	0 (0.0%)	3 (20.9%)	2 (75.6%)	1.4 (35.0%)
	Correct Affinity Estimation	**5 (68.2%)**	**2 (25.4%)**	**0 (0.0%)**	**9 (40.4%)**	**5 (12.1%)**	**4.2 (29.2%)**
	Incorrect Affinity Estimation	0 (0.0%)	2 (21.4%)	0 (0.0%)	1 (3.2%)	1 (0.5%)	0.8 (5.0%)
Model Cannot Be Applied	Total	10 (2.7%)	2 (0.3%)	5 (100.0%)	4 (34.8%)	13 (11.6%)	6.8 (29.9%)
	Too Small (Undefined Affinity)	10 (2.7%)	2 (0.3%)	0 (0.0%)	3 (0.0%)	13 (11.6%)	5.6 (2.9%)
	Insufficient Compile-Time Info	0 (0.0%)	0 (0.0%)	5 (100.0%)	1 (34.8%)	0 (0.0%)	1.2 (27.0%)

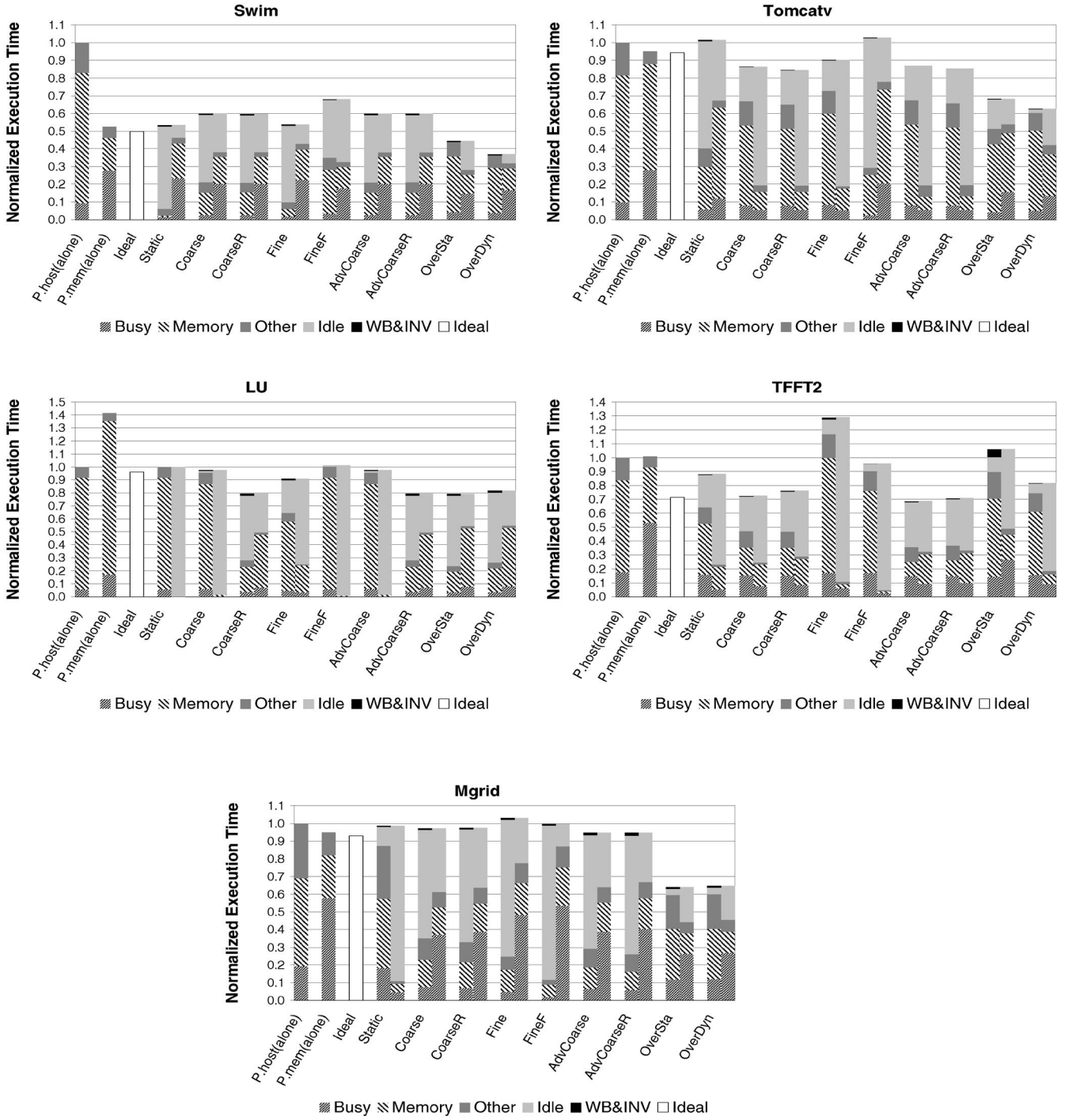

Fig. 3. Execution time of the numerical applications under different scenarios.

5.2.1 Numerical Applications

Fig. 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 nonoverlapped 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 adding up the execution time of each module running 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 overhead. However, we also disregard any dynamic variation of the affinity of any given module because we fix the scheduling of modules to processors. Overall, Ideal is a reasonable approximation to the lower bound of execution time when modules are neither dynamically scheduled nor overlapped.

The remaining, thicker bars correspond to real scenarios where part of the code is executed on P.host and part on P.mem. These bars are thicker because they consist of two component bars glued together: The component on the left corresponds to the execution of P.host, while the one on the right corresponds to the execution of P.mem. Since there is a single program running, both component bars have the same height.

Of these thick bars, we only consider for now those that correspond to nonoverlapped 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 nonideal bars are divided into: execution of instructions (Busy), stall time due to memory accesses (Memory), stall time due to various pipeline hazards (Other), time waiting for the other processor in the machine, namely P.host or P.mem (Idle), and overhead due to write-back and invalidation of P.host caches (WB&INV). The thicker bars show the breakdown of the P.host execution time on the left side and the breakdown of the P.mem execution time 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. TFFT2 runs equally fast on either processor. These results are consistent with the module affinity of the applications shown in Table 8. All together, these results show that neither P.host nor P.mem is the best place to run every single 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 only slightly higher than 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 often as fast as or faster than Ideal. The reason why they can do better than Ideal is that they choose the processor to run each individual module in an adaptive manner. Unfortunately, in the process of doing so, they are likely to run each module suboptimally at least once. This is the reason for the slowdown in Swim.

If we compare Coarse and CoarseR, we see that, in several applications, they perform similarly. This behavior suggests that individual modules hardly change their affinity across invocations. In this case, CoarseR offers little advantage over Coarse.

An obvious exception is LU (Fig. 3). In LU, CoarseR is about 20 percent faster than Coarse. The reason is that several important modules in LU vary their affinity across invocations, allowing CoarseR to adapt. As an example, Fig. 4a shows the workload of one such module as a function of its invocation number. The figure shows the execution time on P.host and on P.mem. We see that the

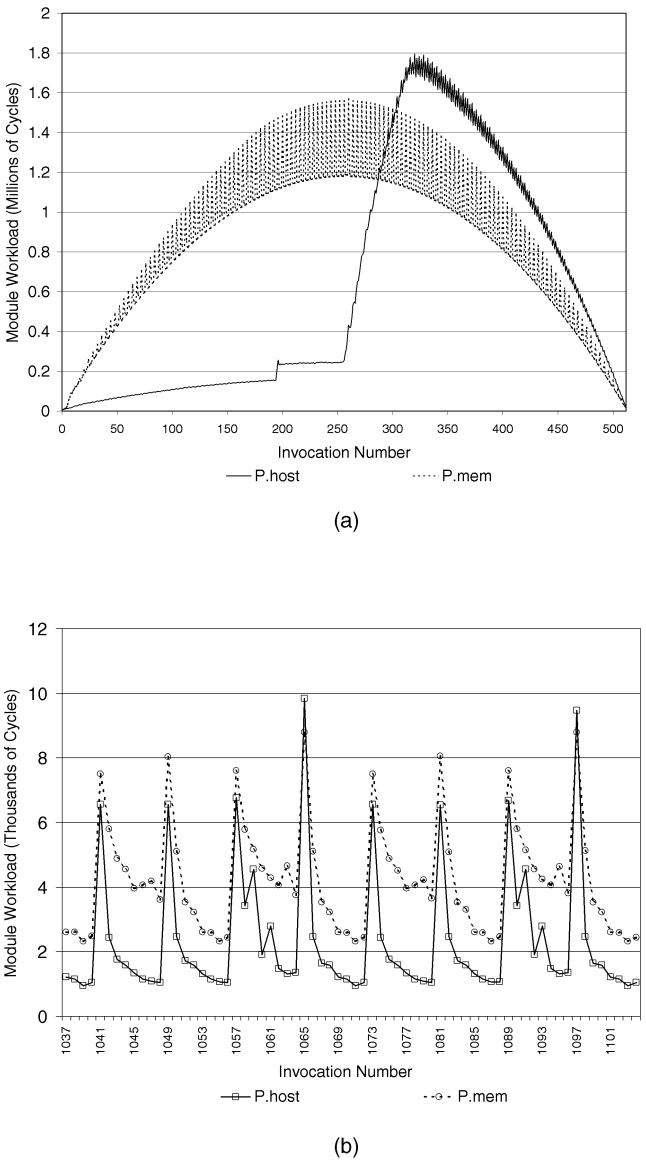

Fig. 4. Variation of workload across invocations in one of the most important modules of LU (a) and TFFT2 (b). In the plots, we show the execution time of the module on P.host and on P.mem.

module has P.host affinity for nearly 300 invocations and P.mem affinity for the remaining 200 invocations. This type of affinity variation allows CoarseR to adapt and execute the module faster than Coarse.

Under some conditions, changes in a module's affinity across invocations may confuse CoarseR and make it slower than Coarse. Such behavior occurs in TFFT2 (Fig. 3). To see how this can happen, Fig. 4b shows the workload of the most significant module in TFFT2 for a window of invocations out of a total of 40,960 invocations. The figure shows that the workload has a periodic behavior and that P.host is nearly always the best processor to run the module. There are only a handful of invocations when P.host is slower than P.mem. As shown in the figure, they correspond to the highest peaks. After CoarseR measures the execution of P.host in one of these peaks, it will schedule

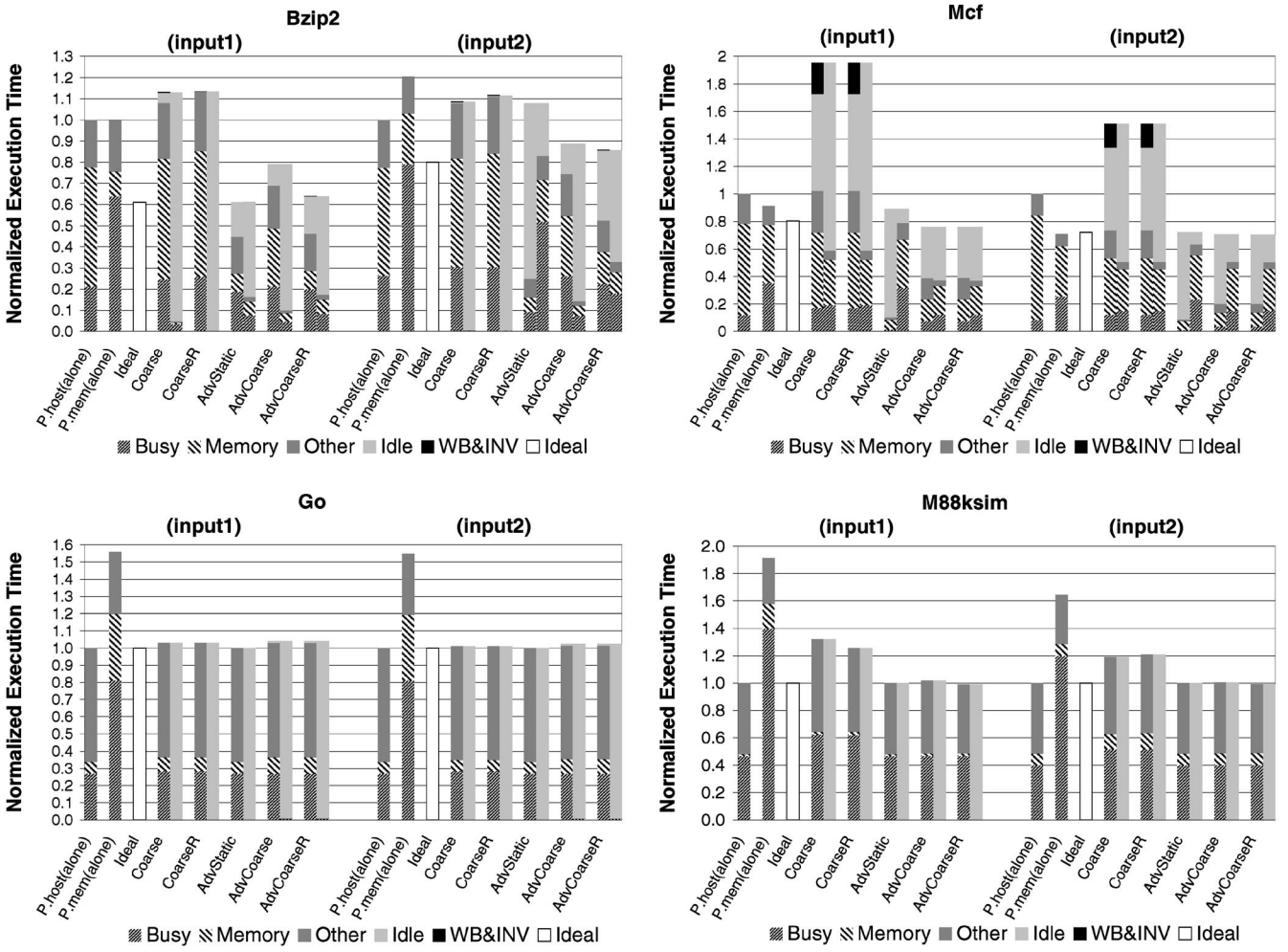

Fig. 5. Execution time of the nonnumerical applications under different scenarios.

the module on P.mem for many subsequent invocations. This effect causes CoarseR to be slower than Coarse.

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 affinity of the module is constant across invocations and iterations, the decision is likely to be suboptimal (Tomcatv).

Overall, we conclude that, for numerical applications, CoarseR is the best strategy under nonoverlapped execution and basic partitioning. CoarseR often compares favorably to Ideal and, on average, it is 20 percent faster than P.host(alone) and 12 percent faster than P.mem(alone). We also conclude that the write-back and invalidation overhead (WB&INV) is negligible.

5.2.2 Nonnumerical Applications

Fig. 5 compares the execution times of the nonnumerical 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 (Table 5). In each application and input set, the bars are normalized and broken down into categories as in Fig. 3. For simplicity, some of the bars in Fig. 3 have been eliminated here.

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 every single application. Go, M88ksim, and Bzip2 for one of the inputs 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 nonnumerical applications is that the Coarse and CoarseR strategies perform poorly now. For example, in Mcf, these strategies take up to 95 percent longer than P.host(alone) to complete execution. The main cause is the small size of basic modules in nonnumerical applications (Table 8). They are so small that the overhead induced by code bundling and instrumentation, module scheduling, and data coherence affects execution time significantly. Consequently, Coarse and CoarseR are not good strategies.

TABLE 10
Characteristics of the Compound Modules Resulting from Advanced Partitioning

Characteristic (% of P.host Time)	Swim	Tomcatv	LU	TFFT2	Mgrid	Average
Total Modules	7 (100.0%)	2 (99.9%)	5 (100.0%)	16 (99.3%)	16 (99.9%)	9.2 (99.8%)
Modules with P.host Affinity	1 (6.5%)	1 (21.6%)	2 (92.6%)	7 (44.0%)	3 (24.0%)	2.8 (37.7%)
Modules with P.mem Affinity	6 (93.5%)	1 (78.3%)	3 (7.4%)	9 (55.3%)	13 (75.9%)	6.4 (62.1%)
Average Number of Invocations	4.6	5.0	409.4	2,564.4	19.1	600.5
Average Module Size (P.host cycles)	24,543 K	35,859 K	1,952 K	2,318 K	1,684 K	13,271 K

Characteristic (% of P.host Time)	Bzip2	Mcf	Go	M88ksim	–	Average
Total Modules	18 (85.4%)	6 (89.3%)	1 (100.0%)	1 (100.0%)	–	6.5 (93.7%)
Modules with P.host Affinity	10 (34.3%)	1 (5.0%)	1 (100.0%)	1 (100.0%)	–	3.3 (59.8%)
Modules with P.mem Affinity	8 (51.1%)	5 (84.3%)	0 (0.0%)	0 (0.0%)	–	3.2 (33.9%)
Average Number of Invocations	2.8	7	1	1	–	2.9
Average Module Size (P.host cycles)	11,264 K	7,576 K	334,998 K	304,695 K	–	164,633 K

For the nonnumerical applications, we use the profiling input data.

5.3 Nonoverlapped and Advanced

We now evaluate the nonoverlapped execution of the applications with advanced partitioning. Again, we consider numerical and nonnumerical applications separately.

In the following discussion, we will refer to Tables 10 and 11. They show the characteristics of the compound modules resulting from advanced partitioning without retraction (Table 10) and with retraction (Table 11). Retraction only needs to be applied to nonnumerical applications. These tables are organized as Table 8.

5.3.1 Numerical Applications

For numerical applications, as we go from basic modules (Table 8) to compound modules (Table 10), we see that the average number of modules per application decreases (from 13.2 to 9.2) and the average module size increases (from 4.6 to 13.3 M cycles). There is virtually no change in the fraction of the application that is covered by the modules. Note that compound modules are still invoked many times. Consequently, there is no need for the retraction algorithm (Section 3.1.3).

With these compound modules, we reexecute the applications under the coarse basic and coarse most recent dynamic scheduling strategies. The result, shown in Fig. 3, is bars AdvCoarse and AdvCoarseR, respectively. Recall that fine strategies are not applicable with compound modules.

If we compare bars AdvCoarse and AdvCoarseR to Coarse and CoarseR in Fig. 3, we see that advanced partitioning improves the average performance of numerical applications slightly: It speeds up TFFT2 and Mgrid by 5-7 percent. There are two reasons for these only modest improvements. First, there is little reuse of data across the basic modules combined into compound modules. Second, the overhead is an insignificant component of the execution time even under basic partitioning. Overall, in spite of the small gains, AdvCoarseR is our best choice of nonoverlapped scheme.

5.3.2 Nonnumerical Applications

By comparing Table 8 with Table 10 for nonnumerical applications, we observe that, on average, advanced partitioning reduces the number of modules per application (from 41.8 to 6.5) and increases their size (from 0.5 to 165 M cycles), while increasing the fraction of the application covered by the modules to 93.7 percent. However, the number of invocations per module is so low (2.9 on average) that we have to apply retraction.

If we apply retraction (Table 11), we see that we increase the average number of invocations per module from 2.9 to 10,923. The cost is a reduction in the average fraction of the application that is covered by modules (from 93.7 percent to 77.0 percent) and a reduction in the average size of the module. For the rest of the experiments in the paper, we use retraction for the nonnumerical applications.

As an aside, we note that, as we go from Table 8 to Table 10, a large fraction of the code in the applications seems to change affinity from P.host to P.mem or vice versa. The reason is that some of the modules in Table 8 are so small that, in practice, we assign them undefined affinity (Section 3.2). These modules can then be combined during advanced partitioning with adjacent modules that have the opposite affinity, creating the effect described.

TABLE 11
Characteristics of the Compound Modules Resulting from Advanced Partitioning with Retraction

Characteristic (% of P.host Time)	Bzip2	Mcf	Go	M88ksim	Average
Total Modules	24 (70.1%)	6 (89.3%)	1 (70.3%)	1 (78.2%)	8.0 (77.0%)
Modules with P.host Affinity	12 (23.3%)	1 (5.0%)	1 (70.3%)	1 (78.2%)	3.8 (44.2%)
Modules with P.mem Affinity	12 (46.8%)	5 (84.3%)	0 (0.0%)	0 (0.0%)	4.2 (32.8%)
Average Number of Invocations	43,581	7	104	1	10,923
Average Module Size (P.host cycles)	3,659 K	7,576 K	2,220 K	81,044 K	23,625 K

Based on the previous discussion, we take the compound modules after retraction and evaluate three strategies: static scheduling, coarse basic dynamic scheduling, and coarse most recent dynamic scheduling. The results are shown in Fig. 5 as bars AdvStatic, AdvCoarse, and AdvCoarseR, respectively.

The figure shows that AdvStatic may perform close to Ideal in some cases. This situation occurs when the production and profiling runs are relatively similar, as in input1 of Bzip2. It also occurs in Go and M88ksim. In these two applications, all modules have P.host affinity for all the input sets evaluated because the working sets fit in the L2 cache. In general, however, AdvStatic is not a good choice because it can perform very poorly, as in input2 of Bzip2.

The adaptability of AdvCoarse and AdvCoarseR to runtime 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. Consequently, AdvCoarseR is our best choice of nonoverlapped scheme. On average, it is about as fast as Ideal, while it runs 12 percent and 30 percent faster than P.host(alone) and P.mem(alone), respectively.

Although not shown in Fig. 5, not using retraction delivers worse results. The AdvCoarse and AdvCoarseR adaptive strategies are not effective. For example, they become as slow as AdvStatic in input2 of Bzip2. Therefore, we recommend to use retraction for nonnumerical applications.

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 the static and dynamic approaches (Section 3.5.1). The result is shown in Fig. 3 as bars OverSta and OverDyn, respectively. We do not show data for the nonnumerical 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 two using loop distribution (Case 2). The remaining seven are not partitioned.

Fig. 3 shows that, for the majority of applications, overlapped execution is significantly faster than the best nonoverlapped strategy, either real (AdvCoarseR) or not (Ideal). Specifically, in Swim, Tomcatv, and Mgrid, the overlapped strategies OverSta and OverDyn are 30-40 percent 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 are no faster than those for nonoverlapped adaptive execution. The reason is that the most significant modules have dependences and, as a result, are neither partitioned nor overlapped.

TFFT2, on the other hand, runs slower because the overlapped strategies induce extra overhead. While some of the overhead comes from the addition of module bundling code and instrumentation, most of the overhead is induced by the need to ensure coherence for the cached data. To ensure coherence, P.host executes instructions to write back and invalidate cached data structures when execution is transferred to P.mem and back (WB&INV in Fig. 3). P.host also executes additional instructions to generate the addresses of these data structures. These additional instructions add to P.host's Busy in Fig. 3, which does not appear to have increased in OverSta because, at the same time, much work has been transferred to P.mem. Finally, P.host suffers additional overhead in the form of cache misses to reload the data in its cache after the cache invalidations (higher Memory stall in Fig. 3).

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

5.5 Overall Speedups

The results in the previous sections have shown which are the best code mapping strategies. If we want to exploit the heterogeneity of the architecture without allowing any execution overlap between processors, the best overall strategy is AdvCoarseR. This strategy should be used with retraction when applied to nonnumerical applications. If, instead, we want to exploit both heterogeneity and execution overlap between P.host and P.mem, the best overall strategy is OverDyn. However, we can only apply this technique successfully on the numerical applications.

To summarize our results, Table 12 compares the speedups for different environments. The first three columns with numbers show the speedups of three different strategies relative to P.host(alone). Recall that P.host(alone) uses the original, uninstrumented code. These three strategies are AdvCoarseR, OverDyn, and an *ideal* system that we call *IdealAmdahl*. *IdealAmdahl* assumes an ideal configuration with two 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 nonnumerical applications, respectively. Then, we compute the *IdealAmdahl* execution time of the applications by taking the P.host(alone) time and dividing by two the time taken by the code marked parallel.

The speedup numbers relative to P.host(alone) are very encouraging. Exploiting heterogeneity without overlap (AdvCoarseR} delivers an average speedup of 1.31 and 1.18 for numerical and nonnumerical applications, respectively. If, in addition, we enable overlap (OverDyn), the average speedup for numerical applications is 1.66. The speedups of individual applications for these strategies are

TABLE 12
Comparison of Speedups for Different Environments

Application		$\frac{P.host(alone)}{AdvCoarseR}$	$\frac{P.host(alone)}{OverDyn}$	$\frac{P.host(alone)}{IdealAmdahl}$	$\frac{P.mem(alone)}{AdvCoarseR}$	$\frac{P.mem(alone)}{OverDyn}$	$\frac{P.mem(alone)}{IdealAmdahl}$
Class	Name						
Numerical	Swim	1.67	2.71	2.00	0.88	1.42	1.93
	Tomcatv	1.17	1.60	1.67	1.11	1.52	1.62
	LU	1.26	1.22	1.04	1.78	1.73	1.01
	TFFT2	1.42	1.22	1.91	1.43	1.24	1.94
	Mgrid	1.05	1.55	1.94	1.00	1.47	2.05
	Average	1.31	1.66	1.71	1.24	1.48	1.71
Non-Numerical	Bzip2	1.37	–	1.01	1.48	–	1.00
	Mcf	1.37	–	1.01	1.15	–	1.01
	Go	0.97	–	1.01	1.45	–	1.00
	M88ksim	1.01	–	1.03	1.79	–	1.02
	Average	1.18	–	1.02	1.47	–	1.01

The speedups of the nonnumerical applications correspond to the average of the runs for the two input sets that are not used for profiling.

comparable and often higher than the speedups for the ideal *IdealAmdahl* execution. The latter delivers an average speedup of only 1.71 and 1.02 for numerical and nonnumerical applications, respectively. We note that the *IdealAmdahl* execution assumes a 2-P.host system that, in addition to being ideal, can be argued to be more expensive than our heterogeneous system.

The poor speedups of *IdealAmdahl* for nonnumerical applications compared to AdvCoarseR show the true potential of exploiting heterogeneity. Furthermore, we expect individual nonnumerical applications to benefit from AdvCoarseR more 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, heterogeneity cannot help.

For completion, the next three columns in Table 12 show speedups relative to P.mem(alone). The strategies evaluated are AdvCoarseR, OverDyn, and *IdealAmdahl* for an ideal machine with two P.mems. As usual, to obtain *IdealAmdahl*, we take the P.mem(alone) time and divide by two the time taken by the code that the Polaris or the Silicon Graphics parallelizing compilers marked as parallelizable.

The speedup numbers show similar, if even more encouraging trends. Specifically, AdvCoarseR is much better than *IdealAmdahl* for the nonnumerical applications. While the ideal two P.mem system may be relatively inexpensive, it is very slow relative to the heterogeneous system. Indeed, Go and M88ksim, which have working sets that fit in P.host's caches, run much faster in the heterogeneous system.

Overall, the results obtained indicate that our heterogeneous architecture is a promising approach to speed up applications cost-effectively. For nonnumerical applications, exploiting heterogeneity alone already delivers speedups that are higher than those in a more expensive and ideal 2-P.host system and much higher than those in an ideal 2-P.mem system. For numerical applications, the combination of heterogeneity and overlap delivers speedups that are comparable to those in the ideal 2-P.host system and only slightly lower than those in the ideal 2-P.mem system.

6 Related Work

Many different types of processing-in-memory or intelligent memory architectures have been proposed. They include research concepts like EXECUBE [13] and successors [14], IRAM [15], Raw [28], Smart Memories [18], Imagine [26], FlexRAM [12], Active Pages [23], and DIVA [8], among others. In addition, there are commercial parts that use this architecture, including Mitsubishi's M32R/D [20] and NeoMagic's MagicGraph [22] chips.

In this paper, we focus on an architecture with host and memory processing. Examples of this architecture are Active Pages [23], FlexRAM [12], and DIVA [5], [8]. Published work on programming these architectures largely assumes that the programmer isolates the code sections to run on the memory processors. In addition, such work has mostly focused on executing sections of code on only a set of identical memory processors. The DIVA researchers are currently automating a solution to the latter problem in a compiler [5]. Our work, instead, tries to automatically partition the code into sections and then schedule each section on its most suitable processor in the heterogeneous machine. Another characteristic of our approach is that it uses a combination of compiler and runtime algorithms to map the code.

Some of the other processing-in-memory architecture projects have ongoing compiler efforts. Typically, their target architecture is quite different than ours. For example, the IRAM project has a compiler effort devoted to automatically generate code for VIRAM, a chip with a simple processor, a vector engine, and memory [11]. The compiler is based on the Cray vectorizing compiler. The goal of the project is to automatically vectorize code for the VIRAM architecture.

The Raw project has a compiler called Maps [1]. The compiler generates code for the Raw chip, which is composed of a set of tiles with both processor and memory. The main problems addressed by Maps are to distribute data across many tiles and to disambiguate memory accesses to specific tiles. Using many tiles enhances memory access parallelism, while disambiguating accesses enables the compiler to manage the communication between tiles more efficiently.

A related compiler project in Raw is Hot Pages [21]. The idea is to use compile and runtime information to perform page-level address translation in software efficiently. By managing data mapping and address translation in software, the system becomes more flexible.

Finally, there is other related work that focuses on exploiting parallel execution in heterogeneous environments like networks of workstations or the Grid. Examples of such work are the Globus [6] and Legion [7] systems. Such work, however, is very different from ours. It uses big module sizes such as whole programs. In addition, it targets highly distributed platforms.

7 Conclusions and Future Work

This paper has presented and evaluated an algorithm to automatically map code on a heterogeneous architecture composed of a host processor and a simpler memory processor. The algorithm carefully partitions the code, schedules the pieces on the most suitable processor, and overlaps the execution of the two processors.

The results obtained indicate that such a heterogeneous architecture can deliver cost-effective speedups. Using a set of standard applications and a software simulator for the architecture, we obtain average speedups of 1.7 for numerical applications and 1.2 for nonnumerical applications over a single host with plain memory. For nonnumerical applications, we exploit the heterogeneity of the processors without overlapping their execution. The resulting speedups are higher than in a more expensive and ideal 2-P.host system and much higher than in an ideal 2-P.mem system. For numerical applications, we exploit both heterogeneity and overlap. The resulting speedups are comparable to those in the expensive and ideal 2-P.host system and only slightly lower than those in the ideal 2-P.mem system.

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 [23], DIVA [8], and FlexRAM [12] can potentially 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

This work was supported in part by the US National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, US Defense Advanced Research Projects Agency Contract DABT63-95-C-0097, Michigan State University, and gifts from IBM and Intel. We thank Paul Feautrier, Michael Huang, David Padua, Jose Renau, the rest of the I-ACOMA group, and the referees 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," *Proc. 26th Ann. Int'l Symp. Computer Architecture,* pp. 4-15, May 1999.

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

[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," *Computer,* vol. 29, no. 12, pp. 78-83, Dec. 1996.

[4] C. Cascaval, L. DeRose, D.A. Padua, and D. Reed, "Compile-Time Based Performance Prediction," *Proc. 12th Int'l Workshop Languages and Compilers for Parallel Computing,* 1999.

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

[6] I. Foster and C. Kesselman, "The Globus Project: A Status Report," *Proc. 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," *Comm. ACM,* vol. 40, no. 1, Jan. 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," *Proc. Supercomputing 1999,* Nov. 1999.

[9] IBM Microelectronics, "Blue Logic SA-27E ASIC," *News and Ideas of IBM Microelectronics,* http://www.chips.ibm.com/news/1999/sa27e, Feb. 1999.

[10] S.S. Iyer and H.L. Kalter, "Embedded DRAM Technology: Opportunities and Challenges," *IEEE Spectrum,* Apr. 1999.

[11] D. Judd, K. Yelick, C. Kozyrakis, D. Martin, and D. Patterson, "Exploiting On-Chip Memory Bandwidth in the VIRAM Compiler," *Proc. Second Workshop Intelligent Memory Systems,* Nov. 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," *Proc. Int'l Conf. Computer Design,* Oct. 1999.

[13] P. Kogge, "The EXECUBE Approach to Massively Parallel Processing," *Proc. 1994 Int'l Conf. Parallel Processing,* Aug. 1994.

[14] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha, "Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies," *Proc. 1996 Frontiers of Massively Parallel Computation Symp.,* 1996.

[15] 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," *Computer,* vol. 30, no. 9, Sept. 1997.

[16] V. Krishnan and J. Torrellas, "An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* Oct. 1998.

[17] J. Lee, Y. Solihin, and J. Torrellas, "Automatically Mapping Code on an Intelligent Memory Architecture," *Proc. Seventh Int'l Symp. High Performance Computer Architecture,* Jan. 2001.

[18] K. Mai, T. Paaske, N. Jayasena, R. Ho, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* June 2000.

[19] R.L. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, 1970.

[20] Mitsubishi, "Embedded RAM," http://www.eram.com, 2001.

[21] C. Moritz, M. Frank, W. Lee, and S. Amarasinghe, "Hot Pages: Software Caching for Raw Microprocessors," Technical Report LCS-TM-599, Massachusetts Inst. of Technology, Aug. 1999.

[22] NeoMagic, "NeoMagic Products," http://www.neomagic.com, 2001.

[23] M. Oskin, F. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* pp. 192-203, June 1998.

[24] D.A. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Technical Report UIUCDCS-R-79-990, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Nov. 1979.

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

[26] 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," *Proc. 31st Int'l Symp. Microarchitecture,* Nov. 1998.

[27] J. Veenstra and R. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," *Proc. Second Int'l Workshop Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS '94)*, pp. 201-207, Jan. 1994.
[28] 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," *Computer*, vol. 30, no. 9, pp. 86-93, Sept. 1997.
[29] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. Addison Wesley, 1991.

Yan Solihin received the BS degree in computer science from the Institut Teknologi Bandung, Indonesia, in 1995, the MASc degree in computer engineering from Nanyang Technological University, Singapore, in 1997, and the MS degree in computer science from the University of Illinois at Urbana-Champaign in 1999. He is currently a PhD candidate at the University of Illinois at Urbana-Champaign. From 1999 to 2000, he was on an internship with the Parallel Architecture and Performance team at Los Alamos National Laboratory. His research interests include high performance computer architectures, intelligent memory systems, and performance modeling. His past works include reconfigurable functional units and image processing tools for forensic document examination. He has authored/coauthored several papers in each area and released several software packages, including Scaltool 1.0, a predictive-diagnostic tool for parallel program scalability bottleneck, at the National Center for Supercomputing Applications (NCSA) in 1999. He was a recipient of AT&T Leadership Award in 1997. He is a student member of the IEEE.

Jaejin Lee received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1999, the MS degree in computer science from Stanford University in 1995, and the BS degree in physics from Seoul National University in 1991. He is an assistant professor in the Computer Science and Engineering Department at Michigan State University, where he has been a faculty member since 2000. He was a recipient of an IBM Cooperative Fellowship and a scholarship from the Korea Foundation for Advanced Studies during his PhD study. His research interests include programming languages and compilers, computer architectures, and systems in high-performance computing. He is a member of the IEEE, IEEE Computer Society, and ACM.

Josep Torrellas received the PhD degree in electrical engineering from Stanford University in 1992. He is an associate professor in the Computer Science Department at the University of Illinois at Urbana-Champaign. He is also vice-chairman of the IEEE Technical Committee on Computer Architecture (TCCA). In 1998, he was on sabbatical at the IBM T.J. Watson Research Center, researching next generation processors and scalable computer architectures. He received an IBM Partnership Award in 1997-2000, a Young Investigator Award and a Research Initiation Award from the US National Science Foundation in 1994-1999 and 1993-1996, respectively, and several awards from the University of Illinois. Dr. Torrellas's primary research interests are new processor, memory, and software technologies and organizations to build uniprocessor and multiprocessor computer architectures. He is the author of more than 75 refereed papers in major journals and conference proceedings. He has been on the organizing committee of many international conferences and workshops. He has taken a leading role in the organization of the series of workshops on scalable shared memory multiprocessors and on computer architecture evaluation using commercial workloads. He is a senior member of the IEEE.