Adaptively Mapping Code in an Intelligent Memory Architecture*

Yan Solihin[1,3], Jaejin Lee[2], and Josep Torrellas[1]

[1] University of Illinois at Urbana-Champaign
[2] Michigan State University
[3] Los Alamos National Laboratory
{solihin,torrella}@cs.uiuc.edu, jlee@cse.msu.edu
http://iacoma.cs.uiuc.edu/flexram

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

Our algorithm is embedded in a compiler and run-time system and maps applications fully automatically using both static and dynamic information. Using a set of applications and a simulated architecture, we show average speedups of 1.7 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 to more advanced PIM systems.

1 Introduction

Processor-in-Memory (PIM) chips, by integrating processor logic and memory in the same chip, enable low-latency and high-bandwidth communication between processor and memory, thereby promising high performance [2, 5, 7, 8, 11, 13, 14, 15, 16, 18, 20]. One interesting use of these chips is to replace the main memory chips in a workstation or server. In this case, PIM chips can act as co-processors in memory that execute code when signaled by the host (main) processor. This approach is taken by Active Pages [13], DIVA [5], and FlexRAM [7] among others.

This class of architectures provide a heterogeneous mix of processors: host and memory processors. Host processors are more powerful, are backed up by

* An extended version of this paper appears in [10]. This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, Michigan State University, and gifts from IBM and Intel.

deep cache hierarchies, and see a higher latency to memory. Memory processors are typically less powerful, see a lower latency to memory, and (at least in theory) are much cheaper. The question we address in this paper is: how to automatically program these architectures?

Previous work on programming these architectures [5, 4, 7, 13] manually identifies the code sections to run on memory processors. This process is not transparent to the programmer. In addition, previous work has largely focused on parallel execution of applications on only the memory processors.

In this paper, we present a compiler and run-time algorithm to automatically partition programs into sections and map each section on its most suitable processor, while maximizing the execution overlap of the host and memory processors. We apply our algorithm to both numerical and integer applications. We find that our algorithm effectively exploits the heterogeneity of the architecture.

2 Intelligent Memory Architecture

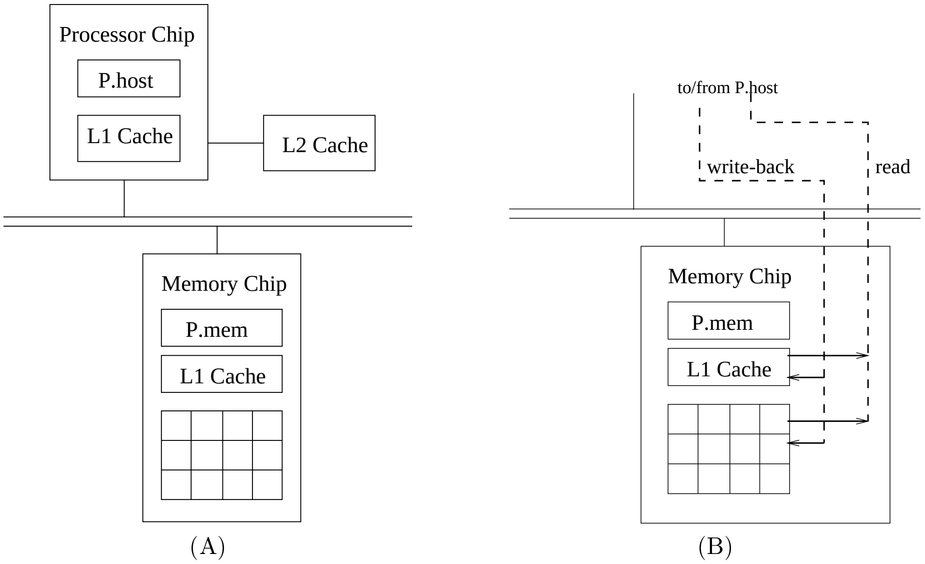

Fig. 1. A simple intelligent memory architecture.

To simplify the problem, in this paper we only consider the simple architecture of Figure 1(A), which has a single host processor (*P.host*) and a single memory processor (*P.mem*). We are in the process of extending our techniques to map code to architectures with multiple processors of each type.

To reduce the cost of the system, the processor and memory chips are connected with an off-the-shelf interconnection. As a result, P.mem cannot be the master of the interconnection to initiate transactions. Furthermore, there is no hardware support for cache coherence between the P.host and P.mem caches. There is, however, some simple support to make programming easier (Figure 1(B)). Specifically, when P.host writes back a line to memory, P.mem's cache is also updated if it contains a copy of the line. In addition, when P.host requests a line from memory, if P.mem's cache has a copy of it, the cache overwrites the data returning to P.host.

With this support, cache coherence can be ensured by the compiler with simple *write-back* and *invalidate* commands that control P.host's caches. The compiler inserts these commands in the program. Specifically, before transferring execution from P.host to P.mem, the P.host cache writes back any dirty lines that the P.mem may want to read. Furthermore, before returning execution to the P.host, the P.host cache invalidates any lines that the P.mem may have written.

3 Modules and Partitioning

Our compiler and run-time algorithm automatically maps both numeric and integer applications to the architecture of Section 2. The algorithm starts by partitioning the code into units of execution called *modules*. Modules have a homogeneous behavior in terms of computing and memory requirements, exhibit good locality, and are easy to extract from the original code. A natural and intuitive unit on which to build modules is a loop.

With *basic partitioning*, we find *basic modules*. A basic module is either a loop nest where each nesting level has only one loop, or a call to a subroutine that is composed of only one such loop nest. The loop nest may span several subroutine levels.

With *advanced partitioning*, we take the basic modules and try to expand their size or to combine several of them while keeping their behavior relatively homogeneous and enhancing locality. The result is *compound modules*. Compound modules do not have to be loops. They are generated by merging basic modules with nearby statements that access overlapping data sets. In addition, they are also generated by combining adjacent basic modules that we expect to have the same *affinity*. We say that a module has affinity for P.host or P.mem if it runs faster on P.host or P.mem, respectively.

To estimate the affinity of a basic module we use two approaches: for integer applications, we use a profiling run with a different input set, while for numeric applications, we use Delphi's static performance predictor [3]. The latter estimates the compute and cache miss times of the module in each processor.

4 Adaptive & Overlapped Execution

The basic or compound modules identified in Section 3 must now be scheduled for execution on either P.host or P.mem. The schedule can be decided statically, based on the affinity estimated by either the static predictor or the profile. This approach we call Static.

A more advanced approach is to decide the schedule adaptively at run time. In this case, the compiler inserts instrumentation code that measures, at run time, the execution time of some invocations of the module (or, if applicable, some of its iterations) on P.host and some on P.mem. Based on these measurements, the run-time system schedules subsequent module invocations (or iterations). The different dynamic scheduling strategies supported are shown in Table 1.

Table 1. Different dynamic scheduling strategies.

Name	What We Do	Note
Coarse Basic (Coarse)	First invocation of module runs on P.host. Second one runs on P.mem. The processor that ran the fastest is assigned the module for the rest of the invocations in the application.	
Coarse Most Recent (CoarseR)	First invocation runs on P.host. Second one runs on P.mem. From now on, after every invocation, we compare the execution time to the most recent execution time on the *other* processor. Based on the result, the subsequent invocation is scheduled on the processor that ran the fastest.	It can adapt to changes in the behavior of the module across invocations.
Fine Basic (Fine)	For each invocation of the module, repeat the following. First iteration runs on P.host. Second one runs on P.mem. The processor that ran the fastest is assigned the rest of the iterations in the invocation.	It only works for modules that have an all-enclosing loop. It may have high overhead.
Fine First Invocation (FineF)	First iteration of first invocation runs on P.host. Second iteration of first invocation runs on P.mem. The processor that ran the fastest is assigned the rest of the iterations in this invocation and the rest of the invocations.	It only works for modules that have an all-enclosing loop. It has the lowest overhead, but it may produce a wrong prediction.

The different strategies in the table may be best under different situations. Table 2 lists the best strategy based on how the behavior of the module execution varies across invocations of the module and across iterations of a given invocation of the module.

Finally, to further speed-up code execution, we overlap the computation of P.host and P.mem. To this end, we divide the application into two classes of regions: *module-wise parallel regions*, where there are multiple modules that can be run in parallel with respect to one another, and *module-wise serial regions*, where only one module can be run at a time because of dependences between modules. Note that here we use basic partitioning because it creates simpler modules and, therefore, exposes more parallelism.

Table 2. Comparing the different dynamic scheduling strategies.

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

In the module-wise parallel regions, we simply overlap the execution of the modules on the different processors. In a module-wise serial region, we try to split the module into two pieces, one for P.host and one for P.mem. Specifically, if the module is a fully-parallel loop, we divide the iteration count into two unequal chunks that are expected to take equal time to execute. Otherwise, we try to use loop distribution across the two processors with or without synchronization. If none of these techniques is possible, we simply apply the best sequential scheduling strategy as described above.

In all cases of overlapped execution, to decide how to partition the work between P.host and P.mem, we can use either static-only information or dynamic information. We call these cases OverSta and OverDyn, respectively.

5 Evaluation Setup

The code generated by the compiler is targeted to a MINT-based [19] simulation environment [9]. The simulation environment can model dynamic superscalar processors with register renaming, branch prediction, and non-blocking memory operations [9]. 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 3 shows the parameters used for each component of the architecture. The L2 cache size used is 1 Mbyte, except for one of the applications (Bzip2), which is simulated with a 512-Kbyte L2.

Our choice of P.mem's clock frequency is motivated by recent advances in Merged Logic DRAM process. They seem to enable the integration of on-chip logic that cycles as fast as in a logic-only chip, with DRAM memory that is only 10% less dense than in a DRAM-only chip [12, 6].

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

Table 3. Parameters of the simulated architecture. Cache and memory latencies correspond to contention-free round-trips from the processor.

Module	Parameter	Value
P.host	Frequency	800 MHz
	Issue Width	Out-of-order 6-issue
	Func. Units	4 Int + 4 FP + 2 Ld/St units
	Pending Ld/St	8/16
	Branch Penalty	4 cycles
P.mem	Frequency	800 MHz
	Issue Width	In-order 2-issue
	Func. Units	2 Int + 2 FP + 1 Ld/St units
	Pending Ld/St	8/8
	Branch Penalty	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 Bzip2), 4-way, 128-B line, 10-cycle hit
	Write-Back Overhead	$5 + 1 \times num\_cache\_lines$ cycles to program. Actual write back of data occurs in background
	Invalidation Overhead	$5 + 1 \times num\_cache\_lines$ cycles total. It is typically overlapped with P.mem's execution
P.mem Cache	L1-Data	Write-back, 16-KB, 2-way, 32-B line, 2-cycle hit
Memory & Bus	Memory Latency	If row buffer miss: 160 cycles from P.host & 21 cycles from P.mem If row buffer hit: 152 cycles from P.host & 13 cycles from P.mem
	Bus Type	Split transaction, 16-B wide
	DRAM Memory Size	64 MB per chip

Assume now that we want to invalidate *num_cache_lines* lines. In this case, P.host suffers a total overhead of $5+1\times num\_cache\_lines$ cycles. Note that these cycles can be overlapped with P.mem execution. This overhead only affects the final execution time if the invalidations have not completed when P.mem finishes execution and P.host is expected to resume.

P.host and P.mem synchronize at module boundaries. Specifically, P.host writes to a register in P.mem to signal P.mem that it can begin execution. When P.mem has completed execution, it writes to another one of its registers so that P.host can see it. The overheads involved in these synchronizations are considered in our simulation.

The applications evaluated are numerical and integer programs: Swim and Mgrid from SPECfp2000, Tomcatv from SPECfp95, LU from [17], TFFT2 from NAS [1], and Bzip2 from SPECint2000. All floating-point applications use double precision. Table 4 shows the data set sizes used and what the applications are computing. For Bzip2, we perform the runs with two different input data sets: the Input1 runs use the Train input set, while the Input2 runs use this paper in postscript format as input. All runs of Bzip2, however, use the profile generated with the Train input set.

Table 4. Applications used.

Application	Data Size and Number of Iterations	Description
Swim	513 × 513, 10 iterations	Shallow water simulation
Tomcatv	513 × 513, 5 iterations	Vectorized mesh generation
LU	512 × 512	LU matrix decomposition
TFFT2	2^{17} elements, 5 iterations	Fast fourier transformation
Mgrid	64 × 64 × 64 grid, 3 iterations	Multi-grid solver: 3D potential field
Bzip2	Input1: Train Input2: This paper in postscript	Compression algorithm

6 Evaluation Results

Table 5 shows the characteristics of the basic modules obtained by our compiler. They account for 97% of the execution time on average. The table also shows that different applications have a different distribution of affinity for P.host and P.mem.

Table 5. Characteristics of the basic modules.

Characteristic (% of P.host Time)	Swim	Tomcatv	LU
Total Modules	16 (100.00%)	7 (96.46%)	5 (99.99%)
Parallel Modules	16 (100.00%)	5 (56.16%)	3 (7.36%)
Serial Modules	-	2 (40.30%)	2 (92.63%)
Modules with P.host Affinity	1 (6.54%)	3 (18.16%)	2 (92.63%)
Modules with P.mem Affinity	15 (93.46%)	4 (78.30%)	3 (7.36%)
Average Invocations per Module	5.7	5.0	409.4

Characteristic (% of P.host Time)	TFFT2	Mgrid	Bzip2 (Input1)
Total Modules	17 (99.28%)	21 (99.79%)	95 (85.37%)
Parallel Modules	15 (93.93%)	18 (96.85%)	28 (0.68%)
Serial Modules	2 (5.35%)	3 (2.94%)	67 (84.69%)
Modules with P.host Affinity	8 (44.04%)	6 (23.91%)	71 (56.41%)
Modules with P.mem Affinity	9 (55.24%)	15 (75.88%)	24 (28.96%)
Average Invocations per Module	1688.5	105.7	48043.0

Figure 2 and Figure 3 show the execution time for each application. In each chart, the two leftmost bars correspond to running the application on P.host alone (P.host(alone)) and on P.mem alone (P.mem(alone)). Then, there are nine pairs of two bars, where each pair corresponds to a different way of partitioning and scheduling. A given pair shows the execution time of P.host and P.mem (which are necessarily the same) broken down into several categories. The pairs of bars correspond to static scheduling (Static), dynamic sequential scheduling with basic modules (Coarse, CoarseR, Fine, FineF), dynamic sequential scheduling with compound modules (AdvCoarse, AdvCoarseR), and overlapped scheduling (OverSta, OverDyn). For Bzip2, since we have two input sets, we only show a subset of the bars.

Non-Overlapped Execution

P.host(alone) and P.mem(alone) show that the relative emphasis on computing and memory activity varies across applications: Swim, Tomcatv, and Mgrid run faster on P.mem, while LU runs faster on P.host and TFFT2 runs equally fast on both processors. Depending on the input set, Bzip2 can be much faster on P.host or almost as fast in P.host as in P.mem. Overall, these bars show that neither P.host nor P.mem is the best place to run all and every application. If an application executes on the less optimal processor, it may take up to 100% longer to run.

Static schedules modules in the floating-point applications according to the static predictor; if the latter cannot estimate the affinity, the module runs on P.host. From the figure, we see that Static performs relatively well. It runs quite fast for Swim, LU, and TFFT2. Overall, Static is attractive because of its simplicity. However, the static predictor does not currently analyze the complicated code in the integer application (Bzip2). Consequently, Static in Bzip2 uses profiling information, which does not necessarily perform well. Specifically, when the input set is very different from the input set used for profiling, Static performs poorly, as it is shown with Input2 in Bzip2.

Coarse and CoarseR tend to be good choices. They are usually as fast or faster than Static. The reason is that they adaptively run the modules on the processors for which the modules have the true affinity. In the process of doing so, however, they are likely to run each module sub-optimally at least once.

Coarse and CoarseR behave similarly for Swim, Tomcatv, Mgrid, and Bzip2 (Input2). In these applications, the workload in a given module tends to remain constant across invocations. As a result, CoarseR does not offer any advantage over Coarse. However, in LU, TFFT2, and Bzip2 (Input1), the workload in a given module varies across invocations. The variation of the modules in LU and Bzip2 (Input1) is gradual, which means that CoarseR can adapt well. The result is that CoarseR is about 20% faster than Coarse in LU. In TFFT2, however, the workload of the largest module varies abruptly, with peaks at every 8 invocations. After one of these abrupt peaks is recorded on a processor, the module is scheduled on the other processor for many subsequent invocations, even though it would run faster on the first processor. This behavior disrupts the smooth execution of the CoarseR algorithm, slowing it down slightly, relative to Coarse.

The fine strategies are not as attractive. Specifically, 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

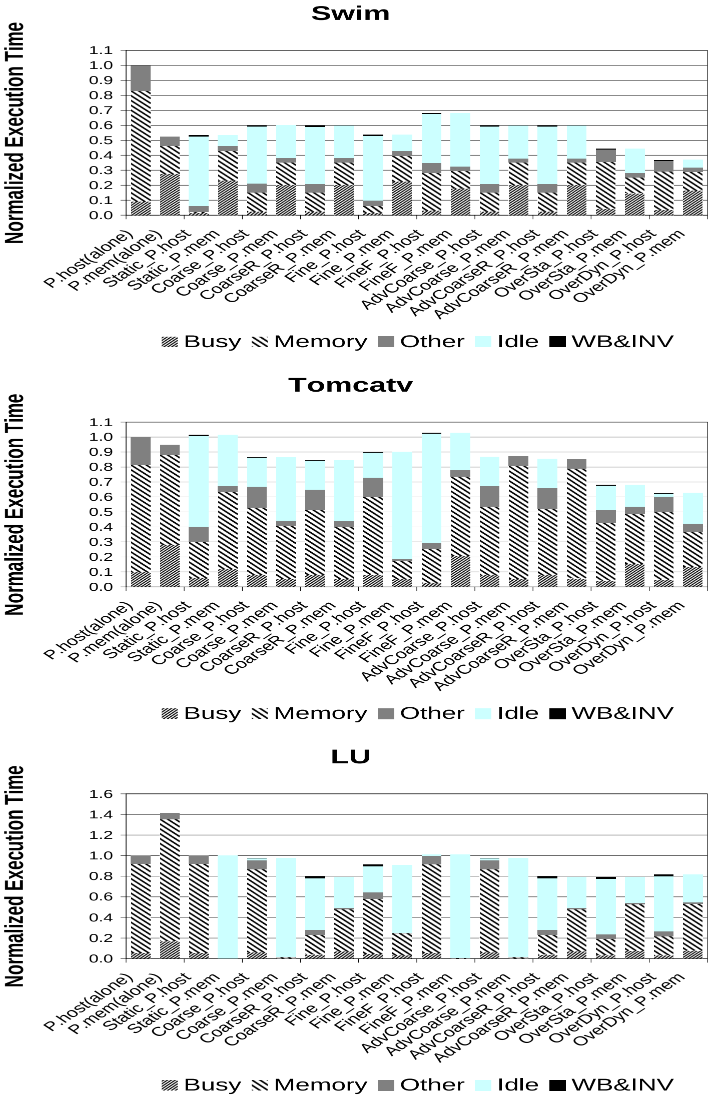

Fig. 2. Execution time of the applications. The bars are divided into execution of instructions (Busy), stall due to memory accesses (Memory), stall due to pipeline hazards (Other), time waiting for the other processor (Idle), and time spent writing back or invalidating lines in the caches of P.host (WB&INV).

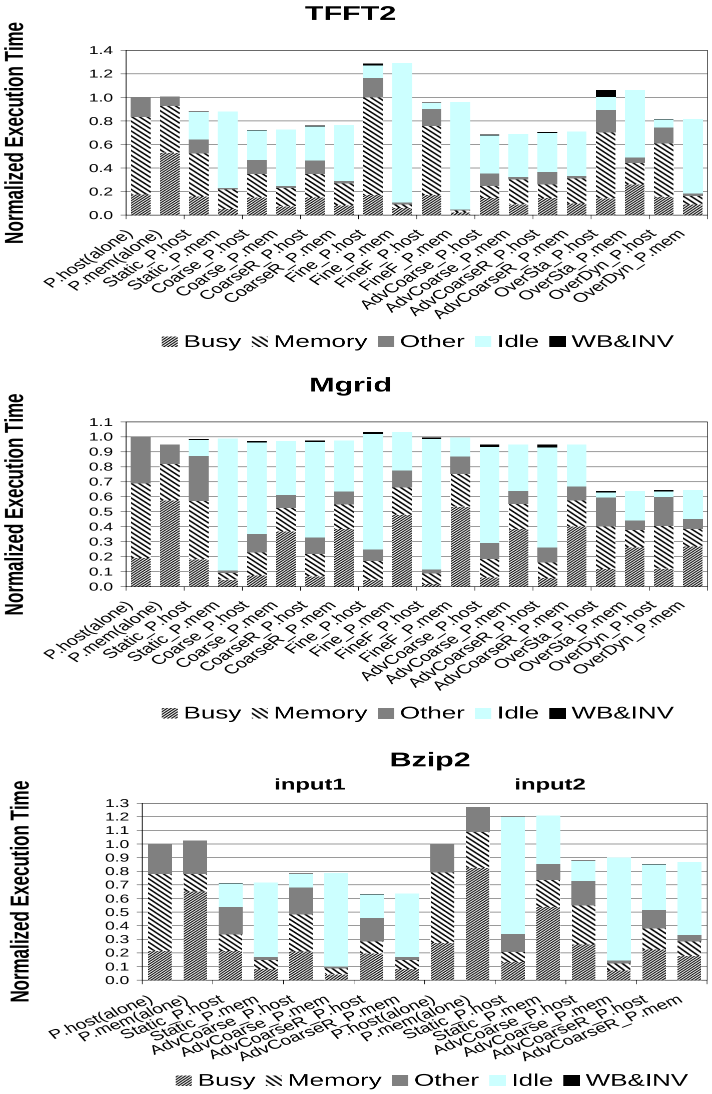

Fig. 3. Execution time of the applications. The bars are divided into execution of instructions (Busy), stall due to memory accesses (Memory), stall due to pipeline hazards (Other), time waiting for the other processor (Idle), and time spent writing back or invalidating lines in the caches of P.host (WB&INV).

first two iterations of the first invocation of the module. Consequently, unless the workload of the module is constant across invocations and iterations, the decision is likely to be sub-optimal (Tomcatv).

The figures show that advanced partitioning has little impact on the performance of numeric applications. Indeed, the AdvCoarse and AdvCoarseR bars are similar to the Coarse and CoarseR ones. The reason is that basic modules are already large enough to dwarf scheduling and other overheads. However, advanced partitioning is effective for Bzip2. Although not shown in the figure, AdvCoarse and AdvCoarseR are significantly faster than Coarse and CoarseR for Bzip2. The reason is that basic modules are quite small in Bzip2 and, therefore, overheads are relatively large.

Overall, we conclude that, among the non-overlapped execution schemes, AdvCoarseR is the best. It is on average 23% faster than P.host(alone) and 20% faster than P.mem(alone).

Overlapped Execution

Overlapped execution speeds up the application significantly in 3 out of 6 applications. Specifically, in Swim, Tomcatv, and Mgrid, the overlapped schemes OverSta and OverDyn speed up the application by 30-40% relative to AdvCoarseR. With these schemes, we are utilizing processor resources that would otherwise remain idle. In LU and Bzip2 (not shown), the overlapped schemes have no impact over AdvCoarseR. The reason is that the most significant modules in these codes have dependences that prevent them from being partitioned following the algorithm of Section 4.

In TFFT2, however, overlapped execution is noticeably slower than AdvCoarseR. The reason is that overlapped scheduling induces extra overheads on P.host. Specifically, it causes extra instruction execution and more cache misses on P.host, all to ensure data coherence. The extra instructions are necessary to write back and invalidate cached data structures when execution is transferred to P.mem and back (WB&INV in Figure 3) and to generate the addresses of these data structures (higher Busy in Figure 3). The extra misses occur when, after the cache invalidations, the data is reloaded into P.host's cache (higher Memory Stall in Figure 3).

Interestingly, the chart for TFFT2 shows that, while OverSta suffers greatly from these overheads, OverDyn is able to eliminate most of them and only takes 12% longer than AdvCoarseR. OverDyn is better because it is aware of the extra overheads involved with overlapped execution and schedules modules more conservatively.

Overall, taking the average over all applications, OverDyn is 18% faster than AdvCoarseR, the best non-overlapped scheme. Consequently, OverDyn is our best scheme of all and, therefore, overlapped execution is our choice.

Summary

As a summary, Table 6 compares the speedups obtained by AdvCoarseR and OverDyn, and the *ideal* Amdahl's speedups for a machine that has 2 P.host processors with plain memory. From the table, we can see that, by running each section of the code on the processor where we expect it to run best, OverDyn delivers an average speedup of 1.6 over a single host with plain memory. The application speedups are very close and often higher than the *ideal* speedups on a more expensive multiprocessor system composed of two identical host processors.

Table 6. Comparing speedups.

Application	$\frac{P.host(alone)}{AdvCoarseR}$	$\frac{P.host(alone)}{OverDyn}$	Ideal Using 2 P.hosts
Swim	1.67	2.71	2.00
Tomcatv	1.17	1.60	1.67
LU	1.26	1.22	1.04
TFFT2	1.42	1.22	1.91
Mgrid	1.05	1.55	1.94
Bzip2 (Input2)	1.17	1.17	1.00
Average	1.29	1.58	1.59

7 Conclusions

We presented and evaluated a compiler and run-time algorithm to automatically partition and map code to a simple intelligent memory architecture. A major conclusion is that some applications run best on the sophisticated P.host, while others run best on the simpler P.mem. Furthermore, the same is true for different code sections within an application. Overall, our results indicate that a heterogeneous mix of processors is a promising approach to speed-up applications cost-effectively. The cost-effectiveness of such architectures is likely to be higher than that of a conventional multiprocessor system with homogeneous processors. We are in the process of extending our algorithms to architectures with several P.hosts and several P.mems.

8 Acknowledgments

We thank D. Padua, A. Hoisie, C. Cascaval, P. Wu, M. Cintra, and anonymous reviewers for useful discussion and feedback on the draft version of this paper. We also thank M. Huang, J. Renau, and C. Cascaval for help with the software used in this study.

References

[1] NAS Parallel Benchmark. http://www.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-98-009/.

[2] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. A. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

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

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

[5] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. In *Supercomputing 1999 (SC99)*, November 1999.

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

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

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

[9] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[10] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code in an Intelligent Memory Architecture. In *International Symposium on High Performance Computer Architecture*, January 2001.

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

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

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

[14] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. Farrens, and A. Chopra. Exploiting ILP in Page-Based Intelligent Memory. In *International Symposium on Microarchitecture*, November 1999.

[15] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick. A Case for Intelligent DRAM. In *IEEE Micro*, pages 33–44, March/April 1997.

[16] D. Patterson and M. Smith. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. 1997.

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

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

[19] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS'94*, pages 201–207, January 1994.

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