The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems*

Richard C. Murphy, Peter M. Kogge, and Arun Rodrigues

Department of Computer Science and Engineering
University of Notre Dame
{rcm,kogge,arodrig6}@cse.nd.edu

Abstract. Processing-In-Memory (PIM) circumvents the von Neumann bottleneck by combining logic and memory (typically DRAM) on a single die. This work examines the memory system parameters for constructing PIM based parallel computers which are capable of meeting the memory access demands of complex programs that exhibit low reuse and non uniform stride accesses. The analysis uses the Data Intensive Systems (DIS) benchmark suite to examine these demanding memory access patterns. The characteristics of such applications are discussed in detail. Simulations demonstrate that PIMs are capable of supporting enough data to be multicomputer nodes. Additionally, the results show that even data intensive code exhibits a large amount of internal spatial locality. A *mobile thread* execution model is presented that takes advantage of the tremendous amount of internal bandwidth available on a given PIM node and the locality exhibited by the application.

1 Introduction and Motivation

Processing-in-Memory (PIM)[14, 13, 4] (also known as Intelligent RAM [21], embedded RAM, or merged logic and memory) systems exploit the tremendous amounts of memory bandwidth available for intra-chip communication, and therefore circumvent the von Neumann bottleneck, by placing logic and memory (typically DRAM) on the same die. This technology allows for the construction of highly distributed systems, but with a very large latency gap between high speed local memory macro accesses and remote accesses. The construction of high performance systems incorporating PIMs must successfully exploit

* An early part of this effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under Cooperative Agreement number F30602-98-2-0180 as part of the Data Intensive Architecture Project under the Data Intensive Program. The U.S. Government is authorized to reproduce and distribute for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

the bandwidth available for on-chip accesses while simultaneously tolerating very long remote access latencies. Multi-threading, similar to that used in the Tera[1], seems the natural method for tolerating remote accesses, however, such a model does not inherently take advantage of the relatively large amount of quickly accessed memory available on a PIM node. In fact, the Tera generally requires about the same amount of persistent state available in the L1 cache of a modern microprocessor[1], and a typical PIM node is likely to have 3 to 4 orders of magnitude more memory available.

This paper describes the memory access behavior of several canonical *data intensive* applications (that is, applications which exhibit frequent data accesses in a highly irregular pattern, and low reuse). These applications, which are particularly difficult for most modern architectures to accommodate, represent scientific problems of significant interest. Thus, the ability to successfully cope with their requirement will yield tremendous insight beyond the more simplistic benchmarks used today.

The characterization of these memory workloads is determined using a **single threaded** trace generated from actual program execution. This represents the first step in modeling a multi-threaded system and identifying a simple data-placement scheme.

This paper is organized as follows: Section 2 describes the benchmarks and the rational for for choosing the Data Intensive Systems suite. Section 3 provides an overview of PIM technology and the general assumptions behind the system simulated. Section 4 enumerates the simulation methodology and describes the desirable outcome of simulation (that is, the condition of success). Section 5 describes the mechanism for analysis, particularly focusing on the Cumulative Instruction Probability Density (CIPD), which will indicate the measure of the degree of success. Section 6 provides the results of experimentation which determine both the size and form of a well constructed working set. Section 7 describes the simulation of a *mobile thread* model of computation in which a thread travels throughout the system looking for the data it needs, as well as the costs and benefits of such a model. Finally, Section 8 contains the conclusions and a description of future work.

Further details on the experimentation described in this paper, as well as a complete set of results for all the benchmarks can be found in [18].

2 Benchmarks

This work concentrates on the analysis of the Data Intensive Systems (DIS) benchmark suite[2, 3]. These benchmarks are atypical in that their memory access patterns exhibit a low degree of reuse and non-linear stride. Thus the focus will naturally be on the performance of the memory system over that of the processing elements. Clearly in the case of PIM the interaction between the demand for data and its supply is the preeminent characteristic under study. Most benchmark suites, in sharp contrast, are designed to be quickly captured in a processor's cache so as to measure raw computation power. This is somewhat misleading since the performance of most modern architectures is determined by that of the memory system.

Early work focused on the performance of the SPEC95[20] integer and floating point benchmarks. The results of those experiments tended to be unenlightening as the memory access patterns were both regular and easily accommodated by even a small PIM (which has significantly more persistent state than modern caches). Tests in which the data set sizes were increased did not fare much better in that the benchmarks themselves tend to use data with a high degree of both spatial and temporal locality.

Significant research was then undertaking using the *oo7* database benchmark[6] with the underlying implementation by Pedro Diniz at USC's Information Sciences Institute, which proved significantly more interesting in that it uses more irregular data structures. Finally, with the release of the DIS suite, which includes a data management benchmark, a sufficient number of distinct data intensive applications were available as a coherent benchmark to allow for meaningful comparison amongst complex applications.

Additional experimentation was performed using a simple Molecular Dynamics simulation[12], which is of significant interest given its highly complex memory access patterns and IBM's Blue Gene project which will use PIM technology for similar protein folding applications. For reasons of brevity, that experimentation will not be summarized here, but can be found in [18].

The DIS suite is composed of the following benchmarks:

– **Data Management:** implements a simplified object-oriented database with an R-Tree indexing scheme [11,16]. Three operations are supported: *insert*, *delete*, and *query*. For the purposes of these experiments, only the *query* operation was examined.
– **FFT:** is a Three Dimensional Fourier Transform which uses the FFTW library for optimization. This operation could have been included as the first step in both the Ray Tracing and Method of Moments benchmarks, however given the code's relatively common use, it is treated separately. (Both the Ray Trancing and Method of Moments benchmarks take data already converted into Fourier space.)
– **Method of Moments:** represents algorithms which are frequency domain techniques for computing electro-magnetic scattering from complex objects. Typical implementations employ direct linear solves, which are highly computation intensive and can only be applied to reasonably low frequency problems. The faster solvers applied in this benchmark are memory bound since reuse is extremely low and access patterns exhibit non-uniform stride. This benchmark is derived from the Boeing implementation of fast iterative solvers for the Helmholtz equation [8,10,9].
– **Image Understanding:** attempts to detect and classify objects within a given image. This implementation requires three phases: morphological filtering, in which a spatial filter is created and applied to remove background clutter; determination of the region of interest; and feature extraction.

– **Ray Tracing:** is a component of Simulated SAR benchmark, and represents the computational core. This portion of the program consists of sending rays from a fixed point and determining where they interact with other objects.

3 PIM Technology and Architecture

Modern processors require that tremendous amounts of data be provided by the system's memory hierarchy, which, is becoming increasingly difficult to supply. The core of this problem, known as the *von Neumann Bottleneck* relates to the separate development of processing and memory technologies, and the different emphasis placed on each. Processors, built around logic fabrication processes which emphasize fast switching, generally follow Moore's law, while memories emphasize high density but relatively low data retrieval rates. The interconnection mechanism between the two is a narrow bus which cannot be greatly expanded due to the physical limit on the number of available pins and high capacitance of inter-chip communication.

Recent developments in VLSI technology, such as the trench capacitor created at IBM, now allow for fabrication facilities which offer both high performance logic and high density DRAM on the same die. These PIMs further allow for the creation of much higher bandwidth interconnection between local memory macros and logic since it all occurs on chip.

Several proposals exist which attempt to fully utilize the potential of these fabrication developments. The IRAM project [21] at Berkeley seeks to place a general purpose core with vector capabilities along with DRAM onto a die for embedded applications. Cellular phones, PDAs, and other devices requiring processing power and relatively small amounts of memory could benefit tremendously from this type of system, even if one only considers the potential advantages in power consumption. Others, such as members of the Galileo group[5] at the University of Wisconsin see PIM as having tremendous potential in standard workstations where the on chip memory macros would become all or part of the memory hierarchy. More recently, the Stanford Smart Memories project[17] began exploring the construction of single chip systems capable of supporting a diverse set of system models.

The DIVA project [13] is currently investigating system and chip level implementations for PIM arrays functioning as part of the memory hierarchy in a standard workstation. Finally, the HTMT[22, 15] project is a multi-institutional effort to construct a machine capable of reaching a petaflop or above in which a large part of the memory hierarchy consists of PIMs being designed by the Notre Dame PIM group. This portion of the memory hierarchy is a huge, two-level, multi-threaded array.

Figure 1 show a typical single node PIM layout. In the case of the target ASAP Architecture[19], a vector processor (capable of operating on 256 bit vectors in 8, 16, or 32 bit chunks) is tightly coupled with a set of memory macros. For the purposes of simulation, it is assumed that the memory macro provides 2 k-bits of data per operation through a single open row register. The ASAP's

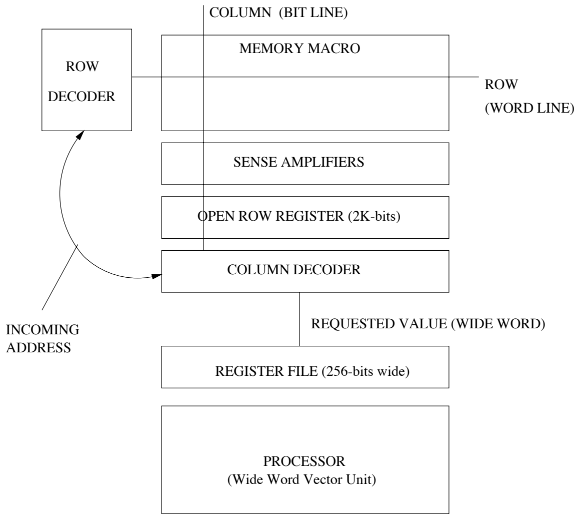

Fig. 1. Typical PIM Memory Layout

register file then accesses that data in 256 bit chunks. Thus, while a random read from memory will cause a DRAM access, a read contained in the current open row does not incur that penalty (because it is simply a register transfer operation).

The array of PIMs simulated is assumed to be homogeneous. Furthermore, for the purposes of this paper, no particular interconnection topology is assume (rather, communication events are merely counted). Experimentation over various topologies can be found in [18]. In actuality, a PIM array is likely to be heterogeneous (potentially consisting of PIMs of different types – SRAM and DRAM – and different sizes), and the interconnection network hierarchical. Multiple nodes may be present on a chip, facilitating significantly faster on-chip communications mechanisms. Additionally, since PIM systems may be part of a larger memory hierarchy, additional non-PIM processing resources or memory may be available.

PIMs, in our model, communicate through the use of *parcels*, which are messages possessing intrinsic meaning directed at named objects. Rather than merely serving as a repository for data, parcels carry distinct high level commands and some of the arguments necessary to fulfill those commands. Low level parcels (which may be handled entirely by hardware) may contain simple memory requests such as: "access the value X and return it to node K." Higher level parcels are more complicated and may take the form "resume execution of procedure Y with the following partially computed result and return the answer to node L." Thus, it should be assumed that parcels can perform both communication and computation, and may be invoked by the user, run-time system, or hardware.

4 Simulation Methodology

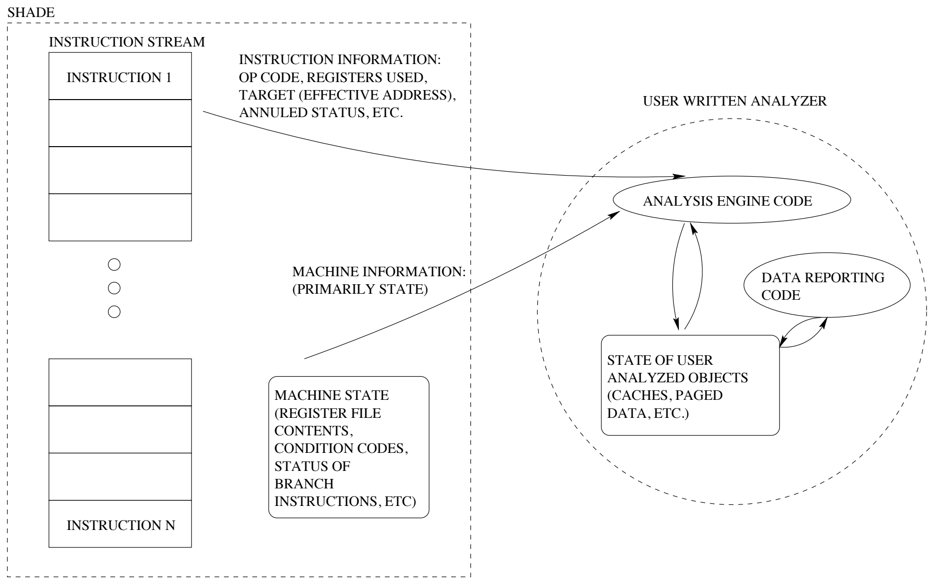
SHADE
INSTRUCTION STREAM
INSTRUCTION 1
INSTRUCTION N
INSTRUCTION INFORMATION: OP CODE, REGISTERS USED, TARGET (EFFECTIVE ADDRESS), ANNULED STATUS, ETC.
MACHINE INFORMATION: (PRIMARILY STATE)
MACHINE STATE (REGISTER FILE CONTENTS, CONDITION CODES, STATUS OF BRANCH INSTRUCTIONS, ETC)
USER WRITTEN ANALYZER
ANALYSIS ENGINE CODE
DATA REPORTING CODE
STATE OF USER ANALYZED OBJECTS (CACHES, PAGED DATA, ETC.)

Fig. 2. Shade Simulations

The principle benchmarking mechanism presented in this paper is the Shade suite[24] developed by Sun Microsystems. This tool allows for the analysis of any SPARC binary by providing a simple mechanism for examining the code's execution instruction by instruction. Figure 2 shows the simulation mechanism. User written analysis code takes the running instruction stream and current machine state to track the state of the processing and memory systems for a PIM array. Of particular interest are memory events, such as opening a new row or generating an off chip memory access.

Since the Shade suite traces SPARC instructions, the simulated ISA corresponds roughly to that of a typical RISC machine. This obviously does not represent the vector ASAP ISA, however, this work is primarily concerned with the performance of the memory system.

Shade does not provide a mechanism for tracing multi-threaded code, though a package to do so is under development and will be incorporated into future work. Consequently the instruction streams analyzed here are **single threaded**. However, since they are taken from the program's main loop of execution, they are not atypical.

To allow the simulation to be tractable, input sets were restricted to the 100-500 MB range, as appropriate for the particular benchmark. Additionally, simulation was limited to a 32-bit address space. Data sets were divided into three parts: code (as indicated by portions of memory subject to an instruction fetch), the stack (which grows down from the top of the address space), and the heap (everything else). For the purposes of data movement, only objects in the heap were analyzed.

Many of the simulations, though consisting of smaller data sets, were performed with an eye towards very large machines (consisting potentially of a million or more nodes). Thus, large parcel sizes (for pages, code, state, etc.), which can be handled by the extremely high bandwidth interconnection networks of such a machine, are not considered detrimental to performance. On the other hand, broadcasts, updating many remote data structures, or overhead data structures which envelop most of the memory on a given node are considered detrimental to performance.

Of particular interest is the amount of time a given thread of execution can continue on a node before an off node memory access is generated. Thus, the execution model favors uninterrupted execution for long periods of time.

5 Metrics

There are primarily two metrics which will be presented throughout the rest of this paper. The first, and simplest to understand, is the *miss rate*. It is, quite simply, the fraction of accesses which cause a miss over the number of accesses during the entire program execution. If A represents the total number of accesses and M represents the total number of misses, the miss rate is merely $\frac{M}{A}$. This is the traditional metric presented when examining the "efficiency" of caches.

However, since the measure of efficiency for the purposes of these experiments is run length between misses (off node accesses), the more detailed *Cumulative Instruction Probability Density*, or CIPD, is also presented. The CIPD is computed by dividing a program's execution up into streams of instructions for which no miss is generated, given the memory state of the machine at the first instruction in each stream. That is, the first instruction encountered which generates a miss constitutes the beginning of the next stream, which means that the previous instruction is the end of the preceding stream.

Streams of the same length (in terms of number of instructions) are placed into buckets. The probability that a randomly selected instruction stream will be from a given bucket is then computed. If the CIPD is represented by the function $\Psi(L)$ where L is an instruction length, $\Psi(L)$ will return the probability that an instruction stream of length greater than or equal to L will be encountered. Thus, for any program, $\Psi(0) = 1$, and if γ represents the maximum length of any instruction stream, $\Psi(\gamma + 1) = 0$. Each of the CIPD graphs which follow represent exactly the function $\Psi(L)$ for each experiment. Ψ can also be used to determine the probability that an instruction stream of length less than or equal to L will be generated. This function, called $\Psi^*(L) = 1 - \Psi(L)$.

It should be noted that the graphs are constructed from individual data points determined during program execution. Since the Ψ always begins at 1 and eventually decays to 0, anything to the left of the beginning of the graph (usually 10^3 instructions) will rapidly reach 1. Similarly, the "end-points" presented are not the true end-points (since they should always become 0); rather they represent the probabilities of the largest instruction streams encountered. Rather than presenting the entire function, these starting and ending points were chosen to better represent the graph and include more information.

There is no notion of weight contained within the CIPD, which can be thought of as "time spent executing." Instruction streams of very long length will show a relatively low CIPD, but could potentially represent the most significant percentage of the overall execution time.

6 Working Set Critical Mass

Of primary concern in the construction of PIM systems is the ability of a PIM to capture a significant working set to perform computation. Modern systems represent working sets in two ways: as a cache or as a page space.

6.1 Caches

Four cache configurations were examined in detail using PIMs of 1, 2, 4, 8, 16 and 32 MB. The configurations were a 256-bit block direct mapped cache, a 2k-bit block direct mapped cache, and 256-bit block 4-way and 8-way set associative caches. (The choice of block size corresponds to assumptions regarding convenient memory access discussed above.) For the purposes of these experiments, only heap data was analyzed (that is, code references were ignored under the assumption that code which is not self modifying can be duplicated across any number of nodes, and stack references were ignored as the size of the active area in the stack tends to be relatively small [18]).

Figure 3 show the typical cache result, in this case using the Method of Moments benchmark. As can be seen from the miss rate, increasing the cache size does not significantly impact the miss rate above cache sizes of 16 MB. Further more, for the most effective configurations (the 256 bit block and 2 k-bit block direct mapped), it does not effect it at all from the initial 1 MB size on. This indicates that temporal locality is exhausted for these benchmarks with a relatively small PIM size. (In this regard, the data management benchmark fared the best, however its best configurations did not improve above 4 MB PIM sizes.) Full simulation details can be found in [18].

Somewhat counter-intuitively, the set associative caches performed worse than the direct mapped configurations. However, given that the caches are so large (as are the block sizes), many sets in both of the set associative configurations remained unfilled. The low reuse of many of the benchmarks further accentuated this outcome.

The increased spatial locality provided by paged memory spaces significantly improved performance. The next section will demonstrate a 1-2 order of magnitude improvement in performance.

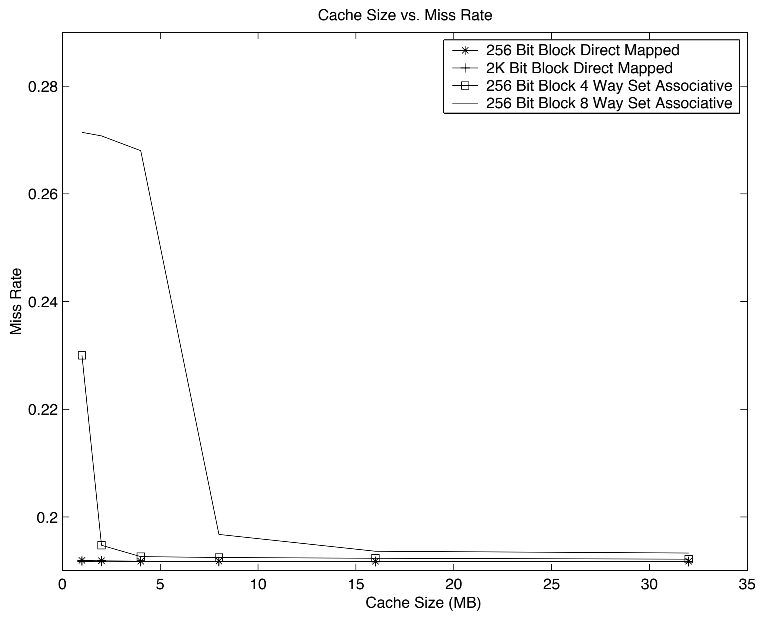

Fig. 3. DIS Method of Moments Cache Size vs. Miss Rate

6.2 Paged Memory

Programmers tend to allocate even pointer-based data in a relatively uniform fashion[23]. This accounts largely for the improvement in performance demonstrated by paged memory configurations. Of primary concern here is the degree to which larger pages are effective, given that on a PIM with a relatively small physical address space, pages which are too large may not allow enough windows into the address space to be effective.

Figure 4 from the DIS Data Management shows the miss rate versus the number of pages on a given node for pages of various sizes (4 KB to 256 KB). The key result given by this graph is that for all PIM sizes tested (1 MB to 32 MB) increasing the page size uniformly improved the miss rate. This indicates that in each case not only was the larger page able to provide additional spatial locality, but having fewer windows into the overall address space did not adversely affect the miss rate.

Obviously larger page sizes place a greater demand on the system's interconnection network during a miss. However, it should be noted that the type of system under examination is assumed to have a very high bandwidth interconnect (with a corresponding high latency for access). Additionally, due to the enormous number of nodes – potentially $O(10^6)$ – possible in such a system, it makes sense to place a greater premium on directory services and the simplicity of name translation than on page transmission time. Finally, no assumptions

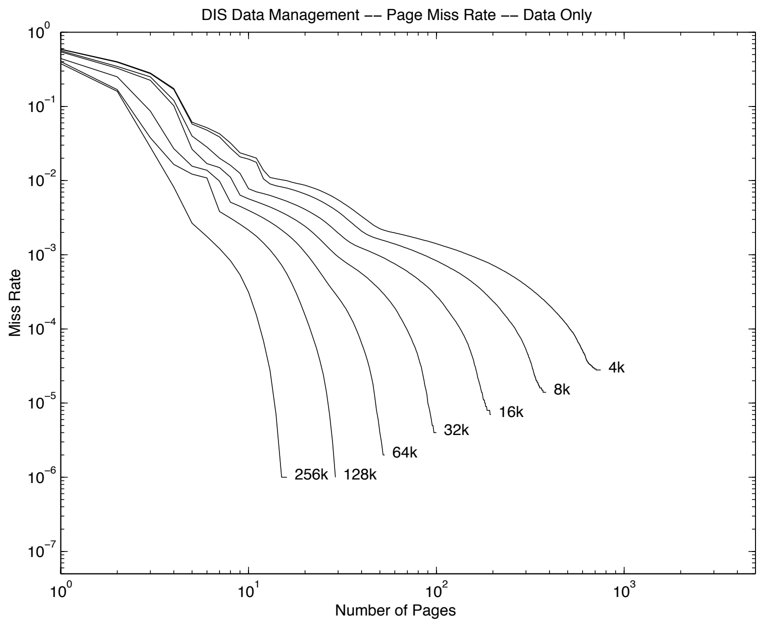

Fig. 4. DIS Data Management Overall Miss Rate

have been made in regard to the location of pages being retrieved (in another area of physical memory, on disk, in a COMA arrangement, etc.), thus assigning a "miss penalty" for the purposes of these experiments is largely irrelevant.

Table 1. Working Set CIPD Mean Values (256 KB pages)

PGM	2 MB	4 MB	8 MB	16 MB	32 MB
DM	10K	13.16M	13.16M	13.16M	13.16M
FF	200	300	600	1.03M	1.89M
MoM	5700	9.26M	2.62M	9.62M	9.62M
IU	632K	655K	9.26M	9.26M	9.26M
RAY	117K	202K	1.11M	1.11M	1.11M

Tables 1 and 2 show the mean and median values of the CIPD ($\Psi(L)$) for each of the benchmarks. They show that relatively small PIMs (4 MB to 8 MB) are highly effective in capturing a working set for most benchmarks. The FFT is a highly unusual case in that 16 MB PIMs are of particular strength in capturing the working set. This is not surprising, however, since the matrices involved are $O(15MB)$ in size.

Table 2. Working Set CIPD Median Values (256 KB pages)

PGM	2 MB	4 MB	8 MB	16 MB	32 MB
DM	2K	20.55M	20.55M	20.55M	20.55M
FF	2K	2K	2K	1.54M	1.72M
MoM	18K	393K	393K	393K	393K
IU	601K	601K	1.53M	1.53M	1.53M
RAY	148K	149K	155K	155K	155K

7 Mobile Threads

Thus far it has been shown that an individual PIM is capable of holding a significant workings set and that increasing the page size significantly improves run lengths on a given node. Furthermore, system design thus far has emphasized not only the long run lengths between remote accesses (due to the relatively low latency of a local memory access versus a remote memory access), but also simplicity in tracking the location of data. In extremely large systems, maintaining a directory of highly fragmented data becomes complex both due to synchronization and storage requirements[18].

Consequently, it becomes increasingly viable to move the computation instead of the data in a *mobile thread* environment. This system, similar to Active Messages[7], extends from the ability of a parcel to invoke computation on a remote node. Under this model, a thread executes until a remote access is generated. At that time, the location of the remote names is determined, and the thread is packaged into a parcel for transmission to the remote node. Upon receipt, the remote node continues the thread's execution.

There are several potential advantages to moving the computation:

– Page tables or other data structures managing the translation of names become small.
– Static data placement significantly reduces the synchronization involved in updating distributed versions of those structures.
– The physical location of a given computation need not be tracked at all. Threads can freely roam the system without causing the update of complicated, distributed data structures. Specifically, if various threads communicate through shared memory, they need not know the physical node upon which the thread with which they are communicating resides, only the location of the shared memory.
– Programming models can emphasize moving to a given node, exhausting the data present, and moving on. Simple mechanisms for delivering such data can easily be provided by the runtime system.
– No round trip communication is necessary since the thread can move to the data rather than requesting data which must then be returned. This eliminated one high latency penalty upon each movement.

Naturally, there are potential problems with such an arrangement:

– Load balancing may be difficult, especially if data placement relies upon highly shared data structures (that is, a given node could become a bottleneck if sufficient computation resources are unavailable).
– The runtime system must be capable of dealing with threads which have run amuck.
– It may be impossible to group data such that related items are together. (This experimentation, using benchmarks which are among the worst known in this regard, indicates that this is really not a problem.)

It is impossible to address all of these problems in this paper, particularly since this experimentation is still in the preliminary phase. Furthermore, the single threaded model adopted for these experiments is incapable of examining contention amongst several mobile threads.

The current model does, however, allow for the characterization of memory access patterns generated by a single mobile thread. Since this single thread represents the main loop of the program, its memory demands should be no smaller than those of its children.

Given the potential difficulties of mobile thread execution, it is likely that a hybrid model will be adopted. For example, data which is heavily shared but not often modified could be duplicated amongst multiple nodes. Additionally, it should be noted that each of the potential problems listed above also occurs with systems which only move data.

7.1 Execution Model

Figure 5 shows the two potential types of mobile thread movement. In the first form, each time a remote memory access is generated a thread is packaged and moved to a new node. A slightly more complex model allows for the thread to communicate with the node upon which it was previously executing in recognition of the fact that some data from that node is probably still necessary during the computation. (This data can, in fact, be captured before the thread moves, which alleviates the reverse communication.)

Data contained on the previous node, if available, is tracked as a "look-back reference." This represents, ideally, what could be packaged up with the thread when it is moved so as to facilitate longer computation on the next node without communication. Of particular interest is the number of unique references to the previous node. Knowing this allows for the construction of data structures to effectively capture such references, and provides a measure of feasibility for mobile threads.

7.2 Data Layout

The experiments to be presented here allow for an extremely simplistic data layout. Heap data is divided into chunks equivalent to a given PIM size, and is held in place. Experimentation in [18] shows that the size of the active stack and

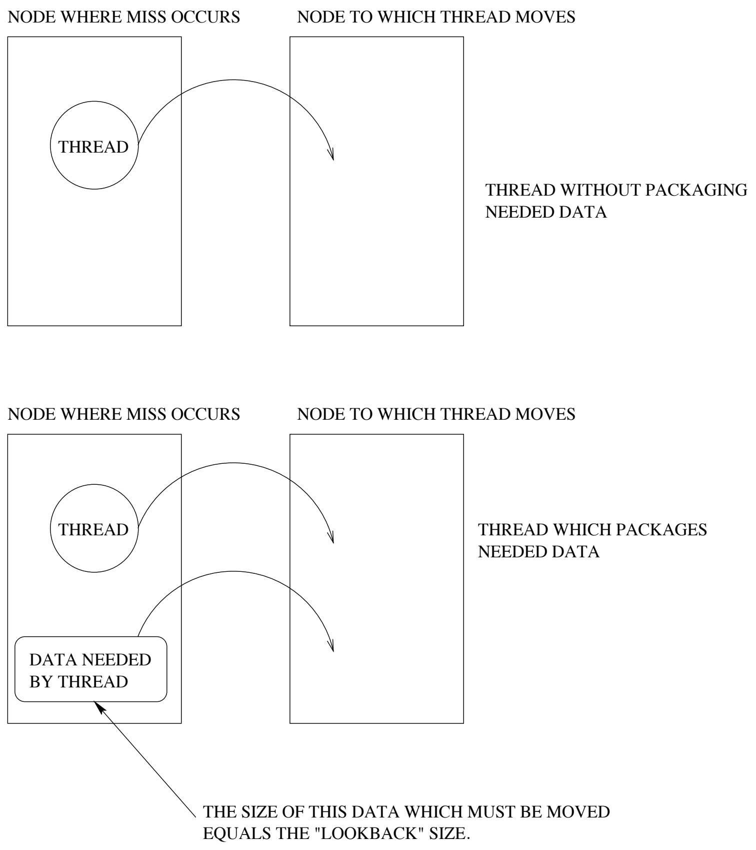
NODE WHERE MISS OCCURS
NODE TO WHICH THREAD MOVES
THREAD
THREAD WITHOUT PACKAGING NEEDED DATA
NODE WHERE MISS OCCURS
NODE TO WHICH THREAD MOVES
THREAD
THREAD WHICH PACKAGES NEEDED DATA
DATA NEEDED BY THREAD
THE SIZE OF THIS DATA WHICH MUST BE MOVED EQUALS THE "LOOKBACK" SIZE.

Fig. 5. Types of Thread Movement

code sections is relatively small (with will over 99% of each being served by in 32 KB of information or less).

The Spartan nature of this data placement tends to yield worst case results. It allows for no compiler, run-time, or user intervention in the policy for placement. Data is merely divided according to PIM size and placed accordingly.

7.3 Run Length Experimentation

Figure 6 shows the impact of backwards references on run length and the overall effectiveness of potential mobile thread computation. In this particular case (DIS Data Management) the results are easiest to understand (and are fairly typical). Because the data structure being traversed is a tree, the PIM size does not significantly alter the run-length data. (The index tree is significantly larger than even the largest PIM studied, therefore in eliminating half of the tree, the thread is required to go to a different node regardless of PIM size.)

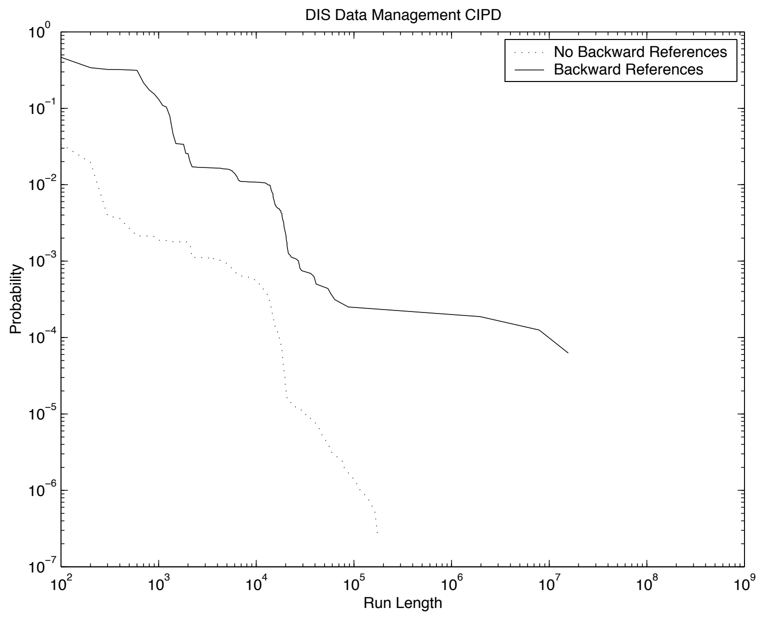

Fig. 6. DIS Data Management Run Length Results (CIPD, $\Psi(L)$)

Allowing for look-back references increased the maximum run length by approximately two orders of magnitude. Similarly, it increased the probability of executing a longer run by nearly an order of magnitude.

Figure 7 shows the most dramatic results. For the Method of Moments code, the maximum run length improved by over four orders of magnitude. Furthermore, not allowing for look-back references yielded particularly bad results – run lengths of over 1,000 instructions occurred less than 0.0001% of the time.

The numerous short run lengths in this benchmark can be attributed to the simplicity of the data placement scheme as related data structures are allocated with very low locality. (Specifically, several big matrices are allocated one after the other, and therefore reside on separate nodes.)

7.4 Look-back Reference Results

Figures 8 and 9 show the probability density of a unique number of references to the previous node being made for a given instruction stream. In every run, except the Image Understanding benchmark, only 10 percent of instruction streams reference more than 10 unique 32-bit words from the previous node, indicating that a very small amount of data is needed to augment a thread once it has moved.

Figure 10 shows the worst case results given by the Image Understanding benchmark. The IU benchmark tended to thrash between the image it was look-

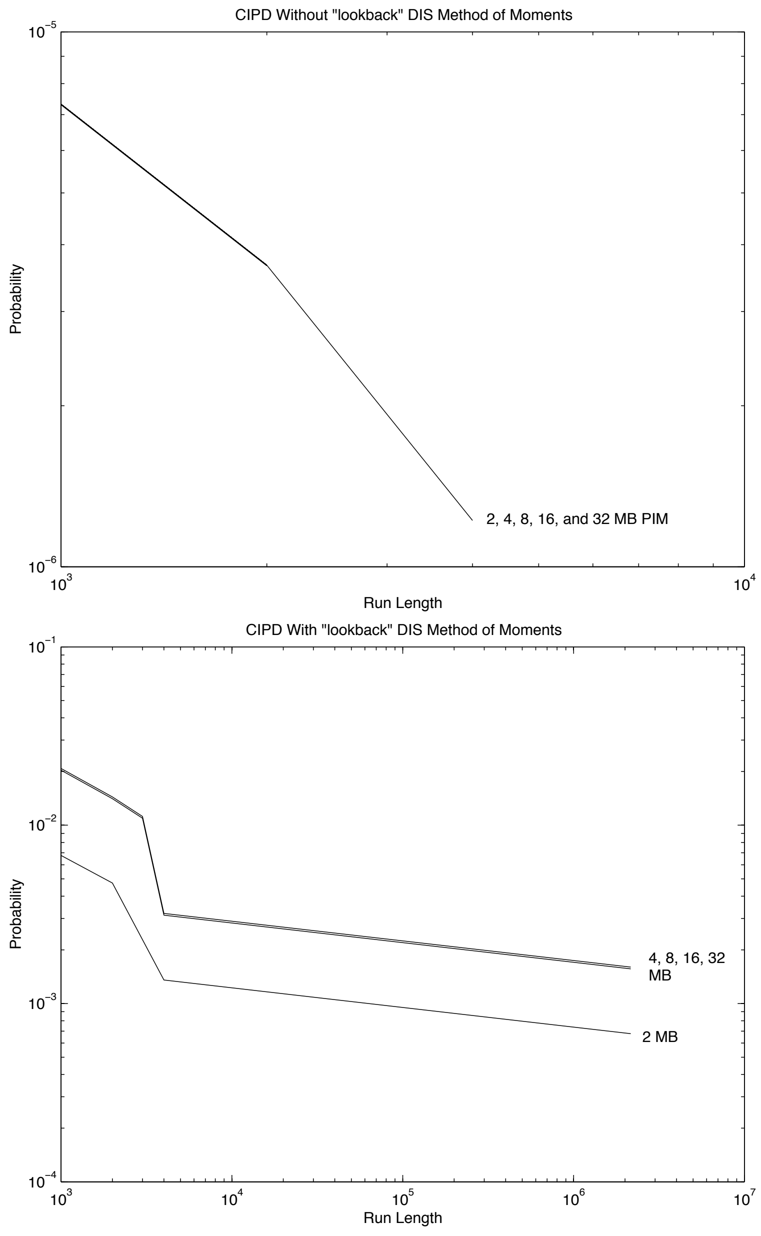

Fig. 7. DIS Method of Moments Results (CIPD, $\Psi(L)$)

ing for and that which it was examining. This problem can be alleviated by better data placement.

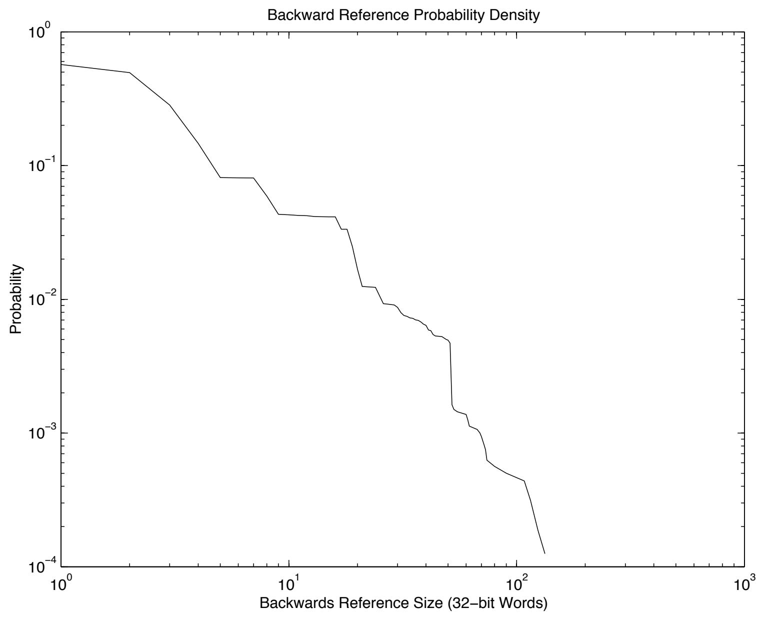

Fig. 8. DIS Data Management Look Back Reference Results

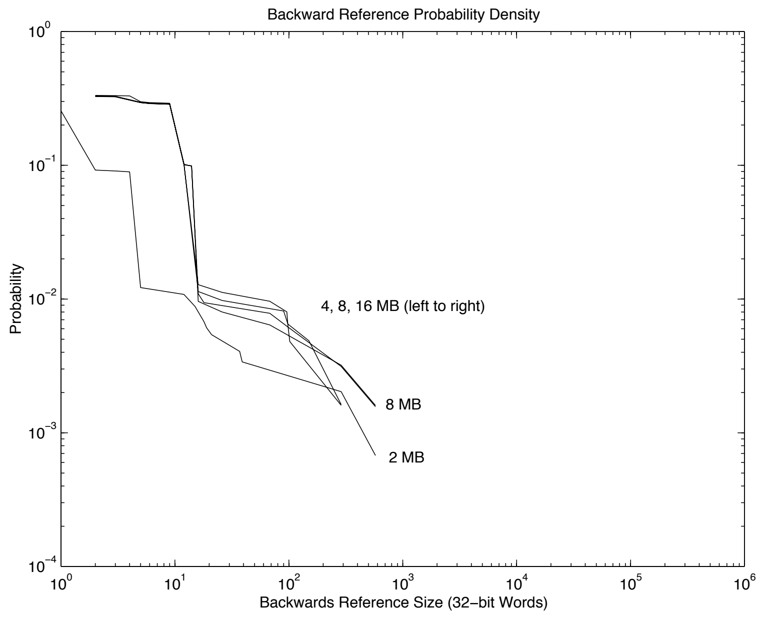

Fig. 9. DIS Method of Moments Look back Reference Results

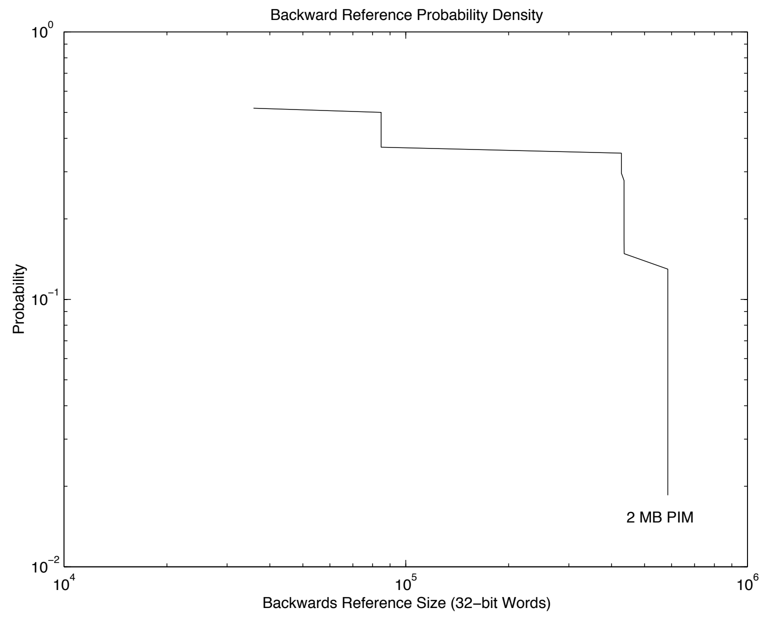

Fig. 10. DIS Image Understanding Look back Reference Results

8 Conclusions and Future Work

This paper examined the architectural parameters which effect program execution on PIM arrays using the Data Intensive Benchmark suite. Because these benchmarks exhibit complex, non-uniform memory request patterns, understanding their characteristics provides an ideal test-bed to flush out the architectural parameters necessary to take advantage of extremely low latency on-node memory accesses. Furthermore, by focusing on applications which have proven themselves difficult for typical memory systems to accommodate, this paper provides a set of "worst (realistic) case" memory access scenarios.

The paramount engineering problem upon which this work centered was the determination of the physical parameters of the design of the memory system (particularly how much physical memory a given node would need to sustain significant computation, and how that memory can be logically organized). While larger memories generally improved performance, it was shown that a relatively small PIM (with a 2 to 8 MB memory macro, for example) can sustain significant computation, and that, in fact, significantly larger PIMs were needed before another order of magnitude increase in executions between misses occurred.

Surprisingly, the increased potential to exploit spatial locality provided by large pages provided significant benefit in all the experimentation. Given that the benchmarks exhibit highly non-linear stride during memory accesses (due to pointer chasing or non-uniform matrix access), and each contained very large data sets, this result, in which fewer windows into the address space are available, took the experimentation in a different direction. Specifically, the number of windows into the address space on each node was reduced to the minimum (one) and the computation was allowed to move between nodes.

Generally, even given the simplistic data placement model, this mechanism proved effective, especially when coupled with the ability to "look back" at the previous node for data which may still be needed. After moving, the amount of data used on the previous node tended to be quite small (on the order of hundreds of bytes), implying that it can be effectively captured and packaged before the thread is moved.

Future work in this area centers upon refining the mobile thread model. A simulator capable of tracking multi-threaded versions of the DIS suite is currently examining the issues of contention, scheduling and traffic. Furthermore, work to define the data structures and hardware necessary to effectively capture look-back references and accelerate the packaging, as well as efforts to define multi-threading constructs capable of supporting inexpensive thread invocations and context switches.

References

1. Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera System.
2. Atlantic Aerospace Electronics Corporation. *Data-Intensive Systems Benchmark Suite Analysis and Specification*, 1.0 edition, June 1999.
3. Atlantic Aerospace Electronics Corporation. *Data Intensive Systems Benchmark Suite,* `http://www.aaec.com/projectweb/dis/`, July 1999.
4. Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A New Memory Semantics for Massively Parallel Computing. In *ICS*, 1999.
5. Doug Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.
6. Michael J. Carey, David J Dewitt, and Jeffery F. Naughton. The OO7 Benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, 1993.
7. David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. February 1995.
8. B Dembart and E.L. Yip. A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels, December 1994.
9. M.A. Epton and B Dembart. Low Frequency Multipole Translation for the Helmholtz Equation, August 1994.
10. M.A. Epton and B Dembart. Multipole Translation Theory for the 3-d Laplace and Helmholtz Equations. *SIAM Journal of Scientific Computing*, 16(4), July 1995.
11. Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOID*, June 1984.
12. J. M. Haile. *Molecular Dynamics Simulation : Elementary Methods.* John Wiley & Sons, 1997.

13. Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.
14. Peter M. Kogge, Jay B. Brockman, and Vincent Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27-28, 1998.
15. Peter M. Kogge, Jay B. Brockman, and Vincent W. Freeh. PIM Architectures to Support Petaflops Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1-3, 1999.
16. Banks Kornacker. High-Concurrency Locking in R-Tree. In *Proceedings of 21st International Conference on Very Large Data Bases*, September 1995.
17. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. *ISCA*, June 2000.
18. Richard C. Murphy. *Design Parameters for Distributed PIM Memory Thesis.* MS CSE Thesis, University of Notre Dame, April 2000.
19. Notre Dame PIM Development Group. *ASAP Principles of Operation*, February 2000.
20. SPEC Open Systems Steering Committee. SPEC Run and Reporting Rules for CPU95 Suites. September 11, 1994.
21. David Patterson, Thomans Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.
22. T. Sterling and L. Bergman. A design analysis of a hybrid technology multithreaded architecture for petaflops scale computation. In *International Conference on Supercomputing, Rhodes, Greece*, June 20-25, 1999.
23. Artour Stoutchinin, José Nelson Amaral, Guang R. Gao, Jim Dehnert, and Suneel Jain. Automatic Prefetching of Induction Pointers for Software Pipelining. *CAPSL Technical Memo, University of Deleware*, November 1999.
24. Sun Microsystems. *Introduction to Shade*, June 1997.