Hierarchical Computing: An Architecture for Efficient Transaction Processing

Juan Rubio and Lizy K. John
Laboratory for Computer Architecture
The University of Texas at Austin
Austin, TX 78712
{*jrubio,ljohn*}*@ece.utexas.edu*

Abstract

Transaction processing workloads impose heavy demands in the memory and storage sub-systems and often result in large amounts of traffic in I/O and memory buses. In this paper, we propose to utilize processing elements distributed across the memory hierarchy, with the objective of performing the computation close to the data residence. Leveraging on active memory modules and active disk devices emerging from other research groups and available in the market, we propose a hierarchical computing system in which the distributed processing elements operate concurrently and communicate using a hierarchical interconnect. Transactions are partitioned across the different layers in the hierarchy depending on the affinity of code to a particular layer or other heuristics. Commands percolate down into the lower layers of the hierarchy and preprocessed/partially processed information flows up into the higher layers. All layers ACTIVELY participate in the processing of the transaction by doing tasks for which they are particularly suited. The lower layers contain inexpensive processor units and in conjunction with the powerful central processor and other collaborating memory and disk processors, yield high performance in a cost-effective fashion. This concept is then applied to the online transaction processing benchmark TPC-C and schemes for code partitioning are outlined. A hierarchical computing system containing four inexpensive memory processors and 32 very inexpensive disk processors can yield speedups of up to $4.52x$ when compared with a traditional system. Since transaction processing has been seen to contain fine-grained and coarse-grained parallelism, the proposed hierarchical computing paradigm which exploits parallelism and reduces data transport requirements seems to be a feasible model for future database servers.

Keywords: Computer Architecture, Parallel Processing, Memory Hierarchy, High Performance Computing, Database Servers.

Technical Area: Architecture

1 Introduction

It is a well known fact that the server market is the driving factor for several of the technological advancements in the computer industry. A few years back, this market was mainly dominated by technical workloads. But during the last two decades, it has changed to power a large portion of commercial operations.

One important type of applications in the group of commercial workloads is Transaction Processing (TP). Transaction processing workloads are classified in two types: Online Transaction Processing (OLTP) and Decision Support Systems (DSS). OLTP systems are used to handle those operations that occur during the normal operation of a business (*e.g.* a client buys products, the managers check the inventory, adjust the price of an item). On the other hand, DSS systems are used to take decisions based on the data gathered by a business, which usually comes from an OLTP system (*e.g.* find most popular product within a given demographic bracket, estimate net profit of all sales in the last three months). Even when both workloads fit within the category of transaction processing, they have many differences:

- OLTP operations are of short duration, taking milliseconds to complete, whereas DSS operations take minutes.
- The above statement also applies to the dataset of an operation. While OLTP operations usually have datasets in the order of kilobytes or megabytes, DSS operations usually access megabytes or hundreds or megabytes of data. Recent literature suggests that DSS systems will be accessing gigabytes in the next couple of years [1].
- The number of concurrent operations in a OLTP system is in the order of thousands, while DSS systems normally have less than a hundred concurrent operations.
- OLTP systems constantly modify the data stored in the databases (*e.g.* enter a sale, deliver a package). DSS systems, on the other hand, use mostly read operations during their execution.

Transaction processing systems are typically implemented using a multi-tier architecture. The idea is to implement a functional pipeline that streams transactions from the clients to the server database in an efficient way. As can be seen from Figure 1, clients on the left are connected to an intermediate server or *Middle-Tier*, through a switched network. The function of the middle-tier server is to act as a filter and reject those requests presented by the clients that are incorrectly generated. It also enforces the security in the system and serves as a parser that transforms requests formulated in one language domain (*e.g.* HTML) to another domain (*e.g.* SQL).

The next component is the Middle-Tier server. In this example, the Middle-Tier server is implemented as a server cluster, with a front-side connection to the clients through a load-balancing switch, whose

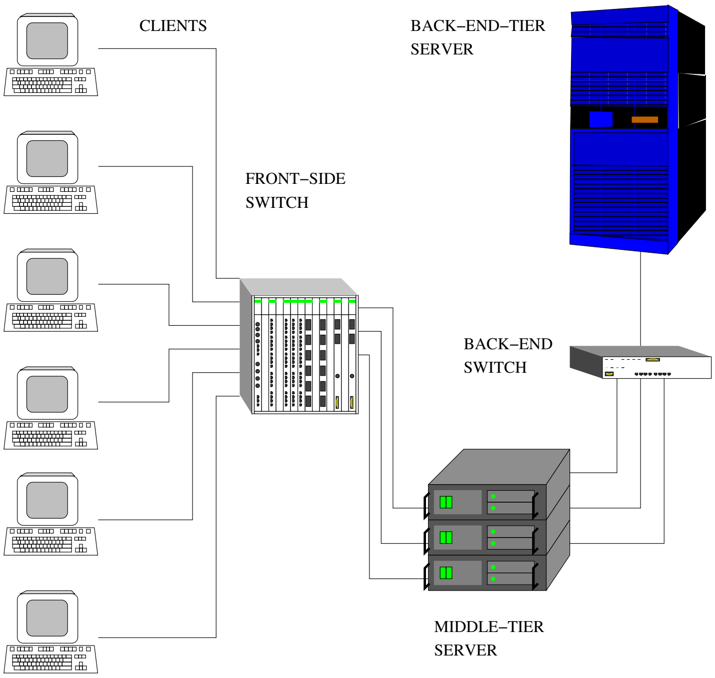

Figure 1: Conventional System Level Architecture for a Transaction Processing System.

function is to create a distributed load in all the nodes of the cluster. Work done in this area includes *locality aware distribution algorithms* like the one developed by Pai, et al.[2]. The nodes in the cluster communicate with each other through a back-side network interface, which they also use to send the requests to the database server, also referred to as the *Back-end-Tier* server.

The final component of the system is the Back-end-Tier, which is also the focus of this paper. This server is the one that manipulates the primary data of the commercial operation (*e.g.* it keeps the list of clients, the orders they place, prices of items and their quantities in the warehouses). As such, the back-end-tier has complete control over a large portion of the data, which is normally local to it and accessed using a Relational Database Manager System (RDBMS or commonly DBMS). Implementations of this server include symmetric multiprocessor systems (SMP) as well as cluster servers.

When we look at the execution behavior of commercial workloads, we observe commercial workloads are different than technical workloads and present more vigorous demands to the memory and storage sub-systems [3, 4, 5]. In fact, studies that analyzed transaction processing workloads indicate that systems spend around 90% of the time waiting for the I/O devices to access the data [6]. Once the data are brought to memory, the processor uses between 25% to 45% of the execution time handling memory accesses [7]. That results in a sub-optimal utilization of the latency hiding features of modern dynamically scheduled processors [8].

One of the reasons for this unbalance between computation and data access backtracks to the principles of traditional memory hierarchies, where data moves from the storage sub-system to the processor before it can be processed. Although we have become accustomed to this execution model, which works well for technical and some other applications, it is far from optimal when used with a transaction processing workload. The action of moving data back and forth between the storage and the computing elements not only results in a high volume of traffic, which hurts the scalability of the system, but it also creates an artificial bottleneck by serializing the execution in an environment with enough parallelism.

This paper presents the Hierarchical Computing model as a possible solution to the problems presented above. The next section introduces the idea, giving special attention to the operation of the hardware and communication of the devices. Section 3 covers the programming model used in the system, how we plan to partition the problem, and use a set of basic primitives to operate on the data. Section 5 performs a mathematical analysis of the idea, and identifies those parameters that affect the performance of this technique. Section 4 presents a basic code partitioning scheme and inspects the idea of using a conventional transaction processing workload, which helps us to determine the feasibility of the model. Section 6 looks at other ideas proposed in the literature. Section 7 concludes with a highlight of the most significant contributions.

2 Hierarchical Computing

To address the problems presented in the previous section, most transaction processing systems exploit the coarse grain parallelism present in the form of concurrent transactions. Before proceeding further it is important to define what is a transaction. A transaction is a sequence of operations, which are executed atomically (atomic), which always maintain a consistent state in the database (consistent), whose execution is not affected by concurrent transactions (isolation) and whose effects are permanent (durable). These set of properties, commonly referred as the *ACID properties* for the first letter of each property, is the basis for the transaction processing theory. Implementations of current systems use a thin layer of software known as the *Transaction Manager*. Its function is to enforce an order in the arrival of the transactions and the ACID properties required by the transaction model. The queries that form a transaction are then passed to the database processes which run in each one of the processors in the system. Although this approach has been relatively successful, it is possible to cause resource management problems, manifested in the form of hot spots during the access of the database tables, thus inhibiting the exploitation of all the parallelism in the system.

The Hierarchical Computing execution model exploits the available parallelism present within a single transaction, in addition to thread-level or coarse level parallelism that can be exploited using other means. The idea is to distribute the computation across a computer system on behalf of a unique transaction. In the model, the communication is based in a message based approach using a hierarchical topology of interconnects. Since this model exploits a different type of parallelism than a traditional system (intra-transaction vs inter-transaction), its use can be orthogonal to existing methods. While in a traditional system all the computations are required to be performed by the CPU, under this model, there is no need to insist on that. There is an important difference between both models, which is seen in the way a server handles a query from a client. In fact, several operations may be efficiently performed at the data residence. This brings a set of interesting tradeoffs, which are presented and evaluated in the next sections.

To distribute computing in this way, some computing power is located in memory and some in disk, close to the location of the data, where they perform simple computations such as comparison and accumulation. The components are coupled using a hierarchical interconnect (any sort of directed acyclic graph like a binary tree).

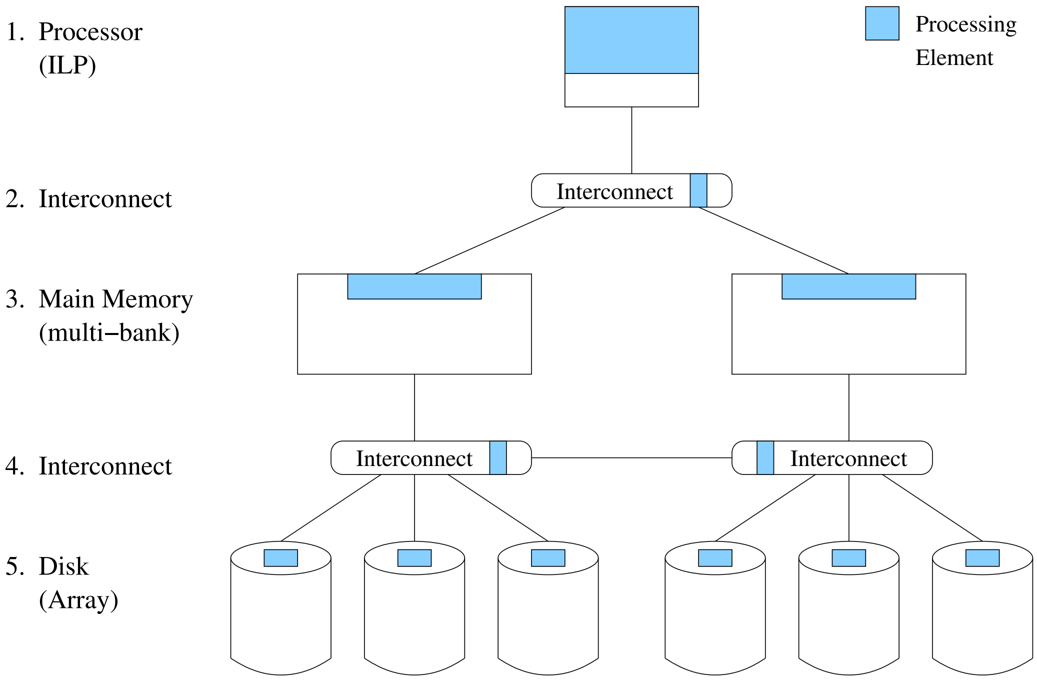

Figure 2: Sample topology for a Hierarchical Computing system.

Figure 2 shows the topology of a sample system based on the above ideas and where we perform computations in five different levels. The additional points of computation are represented by shaded areas and are located in the memory banks, storage devices and interconnects between the main levels. To exemplify the operation of the system, we can introduce the analogy of a corporate office, where the employees are organized in a well defined hierarchy, each of them with an amount of data in its close vicinity and over which they have complete ownership. As a team, they handle each of the transactions received by the office, in a very distributed fashion. Even when at a particular point in time, only some members work in the same task, each one operates on the transaction at some point. Managers at different levels do not need to know every detail of the operations of their subordinates. Each level in the hierarchy conveys only the right amount of information to the layer above.

A system with, say, 1 main memory module and 4 disk modules needs to have only 3 levels of computations (1 in the main CPU, 1 in memory and 1 in the disks), whereas processors in the interconnect will be critical to larger hierarchical computing systems. The number of levels of interconnect processors depends on the number of disks *reporting* to the same memory, as well as the interconnect network, thus for a binary tree we need a number of levels of interconnects expressed by:

$$Max(log_2\ Number\ of\ disks - log_2 Number\ of\ memory\ units - 1, 0) \tag{1}$$

The intelligence in the different levels can be realized using intelligent memory modules investigated in recent research [9, 10, 11, 12], and intelligent or active disks [13, 14, 15, 16, 17]. It is possible to find storage devices in the market with 150 MIPS core and up to 2MB of main memory [18, 19, 20, 21]. Ongoing effort in intelligent or active memories and disk components can thus be leveraged to implement hierarchical computing systems. If computing capability is required in the network or switches, they can be realized using chips similar to micro-controllers embedded in the switch/bus interface. It may be noted that the computing resources required in the storage devices or memory are significantly cheaper than the central processor. Use of a powerful processor which exploits instruction level parallelism and thread-level-parallelism is favored at the root of the hierarchy.

The hierarchical mode of operation has worked relatively well in our society, and we expect it to work as well in a computer system due to the following characteristics:

- Raw data stay local to a level of the hierarchy, which gives more freedom to the upper levels to operate and hold temporary results of the operations in their fast but reduced storage space.
- It suggests a specialization of the units, which permits a system to use dynamically scheduled processors in the upper levels of the hierarchy, where complex decisions are required and control flow is hard to predict. In-order processors, or narrower power-efficient processors can be used to handle the bulk of the data in the lower levels.

The implications of the first point affect the mode of operation of the hierarchy and its programming model and are covered in the next section. The analysis of the second point directly affects the hardware used in the system and will be studied in subsequent sections.

3 Programming Model

As it was mentioned in the previous section, the basic element behind the *Hierarchical Computing* model is the use of computation engines that sit close to the location of the data. The idea is to expedite the movement of data from its natural point of residence to the computation unit. This computation unit can be the main processor, for tasks that require the high computation power provided by a high frequency and dynamically scheduled processor, or could as well be a much simpler microprocessor or microcontroller that functions as an intelligent memory or disk controller. The partition of the data and operations to take advantage of this new architecture are covered in Section 4. This section deals with the programming model used to support a transaction processing workload.

The heart of a transaction processing system is the database server. Nowadays, databases are based on the Relational model [22]. In this model data is stored in the form of tables with a variable number of rows of a predetermined width. To access these tables, these systems use data manipulation languages (DML), of which the *structured query language* (SQL) is the one most frequently used. The operations supported by the SQL language include:

- Scan: locates rows that match a particular criteria. The criteria can be a single predicate (*e.g.* list all flights to Pittsburgh), or a complex predicate (*e.g.* find all vehicles in the state of Texas and registered after 1973). It is called *select* under some data manipulation languages. This operates on a single table.

- Join: similar to the scan operation, but operates on two or more tables (*e.g.* find all vehicles registered before 1966 and currently owned by individuals born after 1966). The two pieces of information are contained in separate tables.

- Insert: creates a new row in an existing table.

- Remove: homologous to insert, it removes a row from a table.

- Update: modifies one or more fields within one or more rows.

- Sort: is not an independent type of operation, but can be applied to a scan operation.

In order to implement the functionality required by the relational model and the SQL language and to take advantage of the concurrency provided by the Hierarchical Computing model, we have opted for a message-based dataflow model. This concept is presented in Figure 3, which shows two levels of a sample hierarchy (although the model is flexible enough to support several levels).

We assume that data is already partitioned and that it resides in the lower level ($L\{1,2\}$). The top level ($T\{1\}$) is the level that initiates the operation by issuing a command (CMD) to one or more modules in the lower levels ($T\{1\}$ and/or $T\{2\}$). The action of sending the command can be a *broadcast* or *multicast* (*e.g.* select all rows which match a criteria), or it can also be a *unicast* (*e.g.* insert row in table). The commands encompass enough information to allow the lower levels to perform the computations in behalf of the upper level. Once the upper level sends the command to the lower level, it waits for data, which depending on the operation can be a null response, a single element or a sequence of elements. The hardware provides basic control flow signals to help the upper level handle the amount of data that might result from an operation.

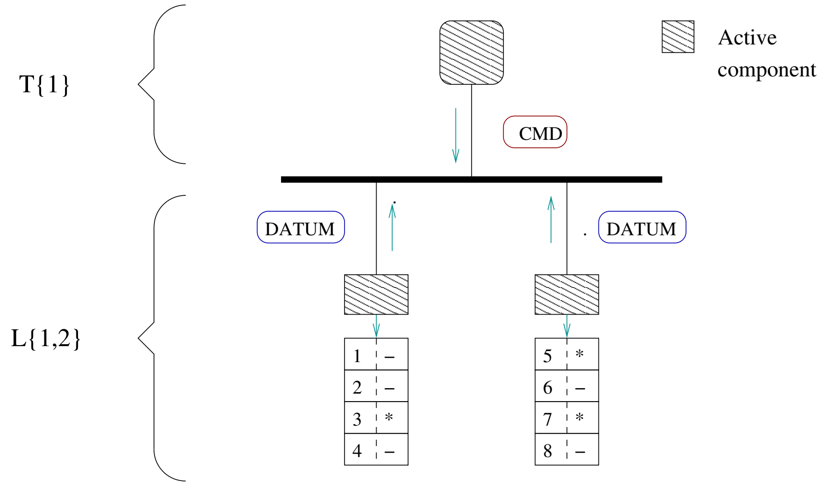

Figure 3: Execution of an operation in the Hierarchical Computing model.

From the perspective of the lower level, once it receives a command from the upper level, it performs a *preorder traversal* starting at this level. Thus, the node proceeds to access the data over which it has control. If the data is not present in its level, it forwards the command to the level immediately under it and forwards all responses to the upper level. If there are no levels under it, a blank response is sent to the upper level.

In order to support this mechanism both commands and responses need to be tagged with a unique identifier. This technique is similar to the tokens present in traditional dataflow machines [23]. We also tag the commands with the *ID* of the level that initiated it, this is done to reduce the overhead of processing the responses. Finally, the model also implements a name space locator in the form of a *table allocation index*. This index permits the processor in the level locate data within its boundaries. It is also used to determine if data is not present, thus avoids a lengthy traversal of all the data.

After studying the different SQL operations and the algorithms used in transaction processing workloads, we designed two types of operations: *individual* and *aggregate*. They are based in the execution presented in Figure 3, but differ in the way the lower level generates the results and how the upper level interprets them.

- Type 1: Individual

  Named *Individual*, as the lower level informs the upper level of every single result. It effectively acts as an unbuffered filter. The semantics can be designed to allow the operation to return on the first event triggered or to continue operating until it reaches the end of the region.

Example: Search a range of data for a string.

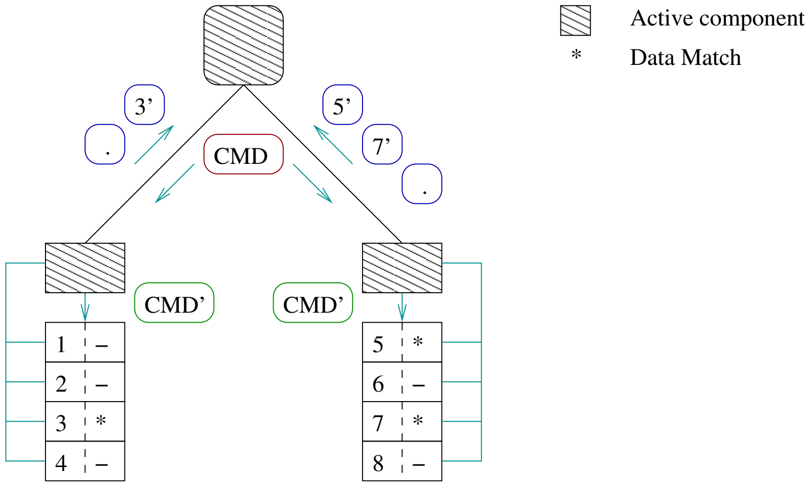

Figure 4: Primitive Type 1 (Individual).

- Type 2: Aggregate

  For this primitive, the lower level accesses its associated data, and finds those elements that match a particular criteria. However, it does not send all these results to the upper level. Instead it produces an aggregate number and sends it once all its data has been analyzed. In this context, an aggregate function $Aggregate()$ is any function that produces a single number based on a set of numbers $(\{I_0, ..., I_N\})$. The most common aggregate functions in transaction processing workloads are: $sum()$, $count()$, $average()$,$max()$, and $min()$. In this case, the results returned by the different nodes in the lower level might need to be combined in the upper level to produce a unique answer. We can accomplish that, if we apply the same aggregate function in the upper level to the values returned. This works for all the examples shown above, except for the $average()$ function. For those circumstances the algorithm is changed to return both $sum()$ and $count()$ from the nodes in the lower levels and the upper level performs the computation of $average()$.

  Example: Count orders placed in January 2001.

4 Code partitioning

When we introduced the programming model in the Section 3, we assumed that the data was laid out correctly in disk. The distribution of code among the different levels of the hierarchy impacts performance. In this section we address the issue of how to partition a transaction in order to achieve a good performance with the Hierarchical Computing model.

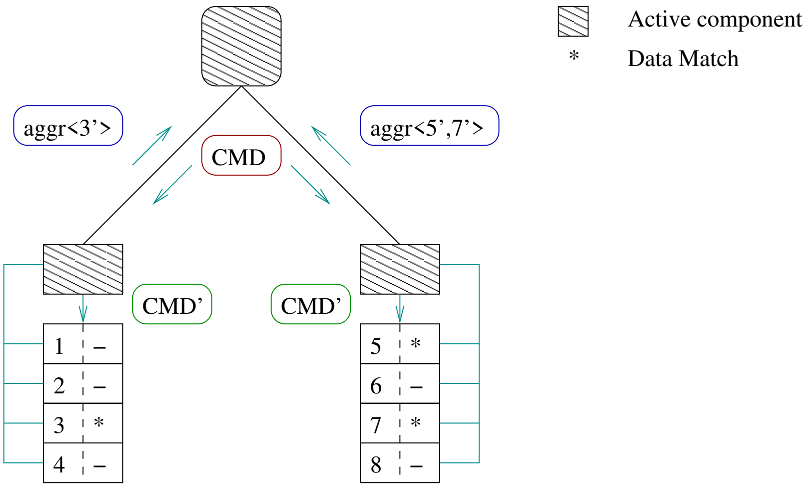

Figure 5: Primitive Type 2 (Aggregate).

To study the data and code partitioning, we begin by looking at the TPC-C benchmark [24], a popular transaction processing workload. The TPC-C benchmark is developed by the Transaction Processing Council (TPC), and is intended to serve as a standard benchmark for Online Transaction Processing systems. The benchmark models the operation of a business with five different types of transactions (new order, payment, order status, delivery and stock level).

Table 1 shows the characteristics of the database tables used by the benchmark. The parameter W represents the number of warehouses in the benchmark, and is used to scale it to different hardware configurations. The cardinality column indicates the number of rows in a database table. The next column shows the size of a row for our implementation using IBM DB2 Universal Database [25]. The last column shows the size of the tables for a configuration with 17,500 warehouses, which is close to the highest non-clustered record to TPC-C by the time we conducted this study.

Specified as part of the TPC-C benchmark is the high level description of each of the five different transactions. For each of these transactions, we show the query execution plans (Figures 6 to 10). Execution plans are directed graphs that represent the tables in the database, the flow of data and the operations performed over the data. The tables are represented by the circles, operations by the rectangles and the flow of data by the arcs.

Note that there is no reference to time in this representation, as the computation is driven by the arrival of data. Also, to read them correctly, they should be traversed in postorder (*i.e.* visit children before entering the node). So if we look at, say, Figure 8, we observe that we cannot perform the *scan* over *table order-line* before we perform the *scan* over *table order*, which itself needs the *scan* over *table customer* together with a *sort* operation. An additional clarification is needed for the transaction shown in Figure 6, where there is a dotted box around some operations. It is used to indicate that a section of the plan is

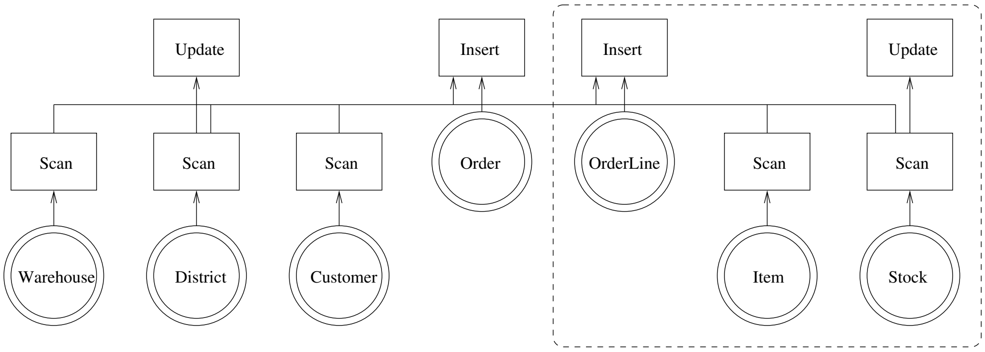

Figure 6: Execution Plan for the TPC-C New-Order Transaction.

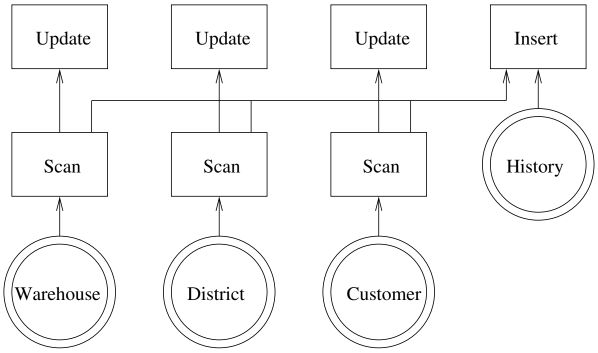

Figure 7: Execution Plan for the TPC-C Payment Transaction.

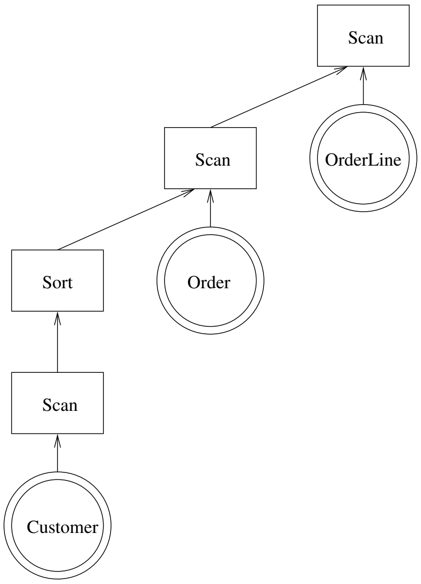

Figure 8: Execution Plan for the TPC-C Order-Status Transaction.

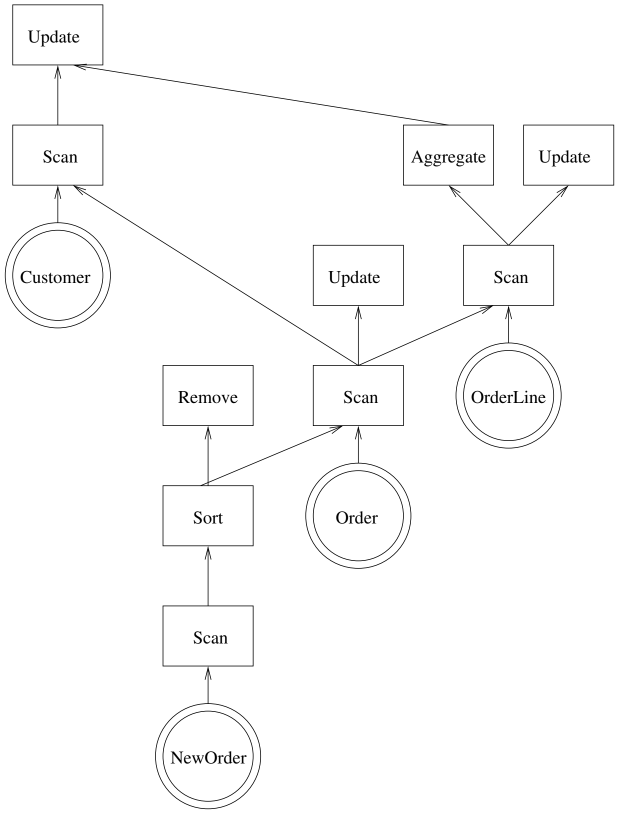

Figure 9: Execution Plan for the TPC-C Delivery Transaction.

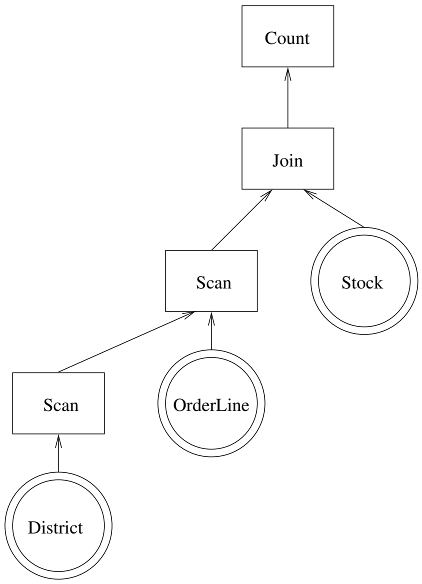

Figure 10: Execution Plan for the TPC-C Stock-Level Transaction.

Table	Cardinality	Row Size (bytes)	Table Size (GB) (W=17.5k)
Warehouse	W	101	<0.1
District	$W \times 10$	107	<0.1
Customer	$W \times 30k$	701	342.8
Stock	$W \times 100k$	330	537.9
Item	$100k$	90	<0.1
Order	$W \times 30k+$	40	19.6
New-Order	$W \times 9k+$	10	1.5
Order-Line	$W \times 300k+$	80	391.2
History	$W \times 30k+$	68	33.3
Total			1,270.4

Table 1: Dimensions of tables for the TPC-C benchmark.

repeated several times. In this particular case, one time for every item in the order.

In a cold system all the data is stored in disk, but as the execution progresses, we can expect tables, a subset of all the rows in the table or indices to move to upper levels in the hierarchy. We call this process *data promotion*. Different from what happens in a traditional system with caches, this operation is not transparent to the software, and is controlled by the mapping algorithms.

Based on the query execution plans, we use a static cost analysis similar to the one used in some of the first query optimizers [26], where the cost (in instructions) of accessing a table is computed as a function of the dimensions of the table and the memory capacity. Given a table $Table_j$ of size $Size(Table_j)$ and with $Rows(Table_j)$ rows, in a hierarchy with M levels, the process used to partition the code is shown in Figure 11. For join operations we use a second table $Table_k$ of size $Size(Table_k)$ and $Row(Table_k)$ rows. This is a simple model, and by no means should be considered an optimal partition. Additional information used by the algorithm might include the notion of processing affinity [27], where a module of computation is sent to the component which will execute it in the minimum amount of time.

5 Evaluation

To evaluate the potential from the hierarchical computing model, we estimate the time needed to perform a task in this model and compare it with the time needed in a conventional system. Equation 2 shows the basic expression for the time necessary to complete a task, where performance gains can be obtained by improving any of the factors of the equation. We have chosen to express the time required to execute an instruction (TPI) as the product of the clock frequency and the CPI (*cycles-per-instruction*). The first is commonly associated with the hardware implementation details, while the second is a factor of the processor architecture and the workload being executed.

Initial table allocation

```
For each Level_i
  For each Table_j
    if (i = M)
      TableInLevel(Table_j, Level_i) ← True
    else
      if (Size(Table_j) < Threshold(Level_i))
        TableInLevel(Table_j, Level_i) ← True
      else
        TableInLevel(Table_j, Level_i) ← False
```

Cost estimation

```
Traverse execution plan
For each operation
  Obtain type(operation)
  Set level where computation is performed
    Levels ← {Level_i | TableInLevel(Table(operation), Level_i) = True}
    HighestLevel ← Min(Levels)
    if (type(operation) = {SCAN, INSERT, REMOVE, UPDATE}
      ExecOpInLevel(operation) ← HighestLevel
    if (type(operation) = {JOIN}
      ExecOpInLevel(operation) ← Min(Rows(Table_j), Rows(Table_k))
      Assist(operation) ← Max(HighestLevel − 1, 1)
  Remember tables used by level
    LevelUsesTable(ExecOpInLevel(operation), TablesUsedBy(operation)) ← True
  Compute cost(operation)
    if (type(operation) = SCAN) cost(operation) ← Rows(Table_j)
    if (type(operation) = INSERT) cost(operation) ← 1
    if (type(operation) = REMOVE) cost(operation) ← 1
    if (type(operation) = UPDATE)
      if (dependencies(operation) > 0) cost(operation) ← Rows(Table_j)
      else cost(operation) ← 0
    if (type(operation) = AGGREGATE) cost(operation) ← 0
    if (type(operation) = SORT) cost(operation) ← Rows(Table_j)
    if (type(operation) = JOIN) cost(operation) ← Max(j, k)
  Add cost(operation) to CostOfLevel(ExecOpInLevel(operation))
```

Remove unused copies of tables

```
For each Level_i where i > M
  For each Table_j
    if (LevelUsesTable(Level_i, Table_j) = False)
      TableInLevel(Level_i, Table_j) ← False
```

Figure 11: Simple scheme for partitioning transaction processing workloads.

$$t = \frac{Instructions \times TPI}{N_{processors}} \tag{2}$$

$$TPI = clk \times CPI \tag{3}$$

This execution model performs computations in the different levels of the storage hierarchy, in what can be considered heterogeneous computing elements, thus Equation 4 expresses the time spent computing by a single level. The index level i goes from 1 to M, where M is the total number of levels in the hierarchy. [1] The parameter N_i indicates the number of processing elements in the particular level i.

$$t_i = \frac{Instructions_i \times TPI_i}{N_i} \tag{4}$$

Another interesting characteristic of this model, is that it allows for operations to be performed concurrently in the different levels. However there might be algorithms that do not allow for that level of parallelism because they have serial components in their execution. Taking that into account, Equations 5 and 6 show the range of values for the total time taken by a task. When we can exploit full parallelism, the resulting time will be the maximum of the individual times. Situations where all the operations need to be serialized would result in a time equal to the sum of all the times. Hence the range of execution time can be calculated as follows:

$$t_{max} = \sum_{i=1}^{M} t_i \qquad \textit{no overlapping} \tag{5}$$

$$t_{min} = \underset{i=1}{\overset{M}{MAX}}\ t_i \qquad \textit{full overlapping} \tag{6}$$

To enable this model, computation is partitioned so operations are assigned to the computation elements in each one of the levels. Considering the nature of transaction processing workloads, the partition of operations can be done statically or using query optimizers like the ones built in most commercial databases [26]. With these systems, we express the distribution of computation in Equation 7. The coefficients φ_i indicate the fraction of the total number of instructions executed in level i of the hierarchy.

$$Instructions = \sum_{i=1}^{M} Instructions_i \tag{7}$$

$$\varphi_i = \frac{Instructions_i}{Instructions} \tag{8}$$

[1]Our convention is to number the hierarchy from the top to the bottom, so the level with the main processor is assigned a value $i = 1$.

The number of instructions a processor is capable of executing is a function of a myriad of factors in the system. Among them, the type of workload plays an important role. Given that we are evaluating this model under the light of transaction processing workloads, we will use some

$$CPI = CPI_{computation} + CPI_{storage} \tag{9}$$

A characteristic of this model, is the use of relatively simple computation engines in the lower levels of the hierarchy. This affects the balance of the operations in the system, according to Equation 10.

$$CPI_{computation,i} < CPI_{computation,j} \qquad i < j \tag{10}$$

Assuming a hierarchy of three levels like the one shown in Figure 12, it is possible to present an expansion of the above relations, showing the factors in which the time decomposes. We have assigned the first level to operations computed by the main processor, the second level to the main memory and the third level to those operations performed by the disk controller.

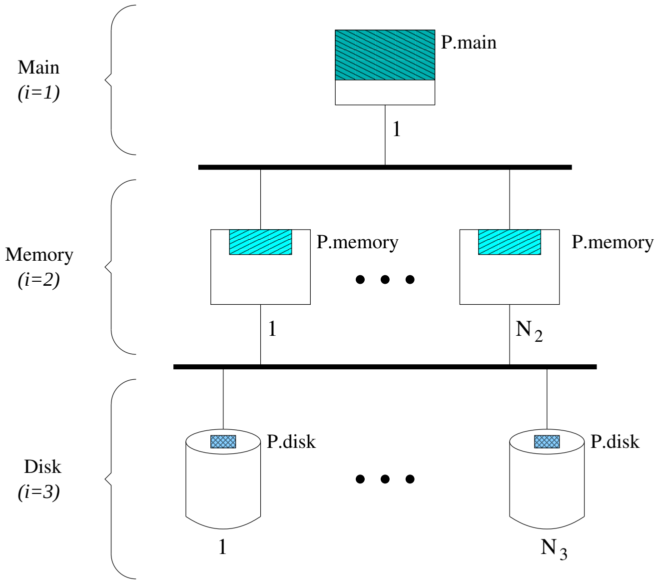

Figure 12: Topology of a Simple Hierarchical Computing system with $M = 3$.

$$t = \underbrace{\frac{Instructions_1 \times TPI_1}{N_1}}_{main} + \underbrace{\frac{Instructions_2 \times TPI_2}{N_2}}_{memory} + \underbrace{\frac{Instructions_3 \times TPI_3}{N_3}}_{disk} \tag{11}$$

Since typically the disk and memory processors will not be as sophisticated as the central processor, we use a degradation factor to indicate how the memory and disks processors compare to the central processor.

The degradation factor for TPI_i is designed as $\tau_i = \frac{TPI_i}{TPI_1}$, which we use to obtain the speedup for the non-overlapping mode using Equation 5 is:

$$Speedup = \frac{1}{\varphi_1 \; + \; \varphi_2 \; \tau_2 \; \frac{N_1}{N_2} \; + \; \varphi_3 \; \tau_3 \; \frac{N_1}{N_3}} \tag{12}$$

Similarly, the speedup when the algorithm shows full overlapping is expressed as:

$$Speedup = \frac{1}{MAX(\varphi_1, \; \varphi_2 \; \tau_2 \; \frac{N_1}{N_2}, \; \varphi_3 \; \tau_3 \; \frac{N_1}{N_3})} \tag{13}$$

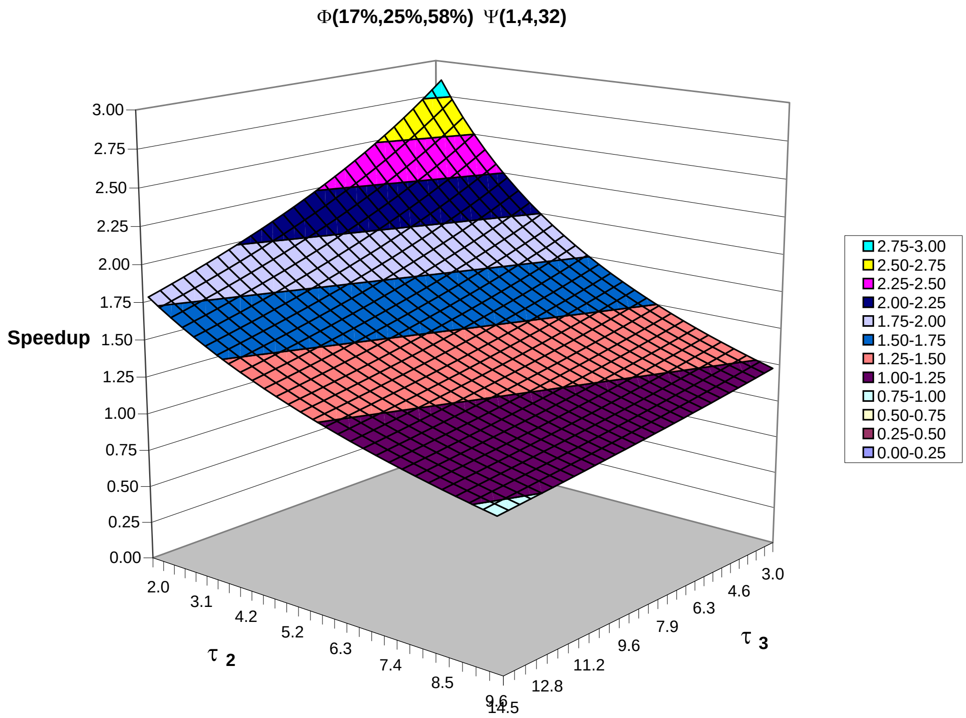

Figure 13: Speedup for $\Phi(17\%, 25\%, 58\%)$ $\Psi(1, 4, 32)$.

Let us use tuple Ψ to represent the number of computation elements in each level of the hierarchy. Code will be partitioned between the different layers based on affinity or other heuristics. We use tuple Φ to represent the fraction of the instructions executed in each level of the hierarchy (φ_i). To analyze the effect of the selection of devices, we generate the speedup surface for variations of τ_2 and τ_3 (Figure 13). As it was expected, we achieve the maximum speedup when we use the smallest numbers for τ_2 and τ_3 (*i.e.* P.memory and P.disk are not significantly slower than P.main). There are however several points with equal speedup for multiple pairs of degradation factors, which gives designers a great degree of freedom when selecting components.

Ψ	Φ	Speedup Min	Speedup Max
(1, 4,32)	(10%, 5%, 85%)	1.60	4.32
(1, 4,32)	(35%, 10%, 55%)	1.17	2.13
(1, 4,32)	(15%, 25%, 60%)	0.95	2.86
(1, 4,32)	(0%, 25%, 75%)	1.01	4.52
(1, 4,32)	(10%, 20%, 70%)	1.08	3.48
(1, 4,16)	(10%, 20%, 70%)	0.80	2.67
(1, 4,32)	(10%, 20%, 70%)	1.08	3.48
(1, 4,64)	(10%, 20%, 70%)	1.31	4.10
(1, 2,32)	(10%, 20%, 70%)	0.70	2.58
(1, 2,64)	(10%, 20%, 70%)	0.79	2.91

Table 2: Speedups of hierarchical computing system with conventional memory and disk components.

The algorithm presented in Figure 11 is used to obtain a preliminary estimate of the code partitioning. For the situation where $Threshold(Level_i) = Capacity(Level_i)$, Table 3 shows the fractions of the computation that should be performed in each level for a configuration of $\Psi(1, 4, 16)$.

Transaction type	φ_1	φ_2	φ_3
New Order	10.0 %	0.8 %	89.2 %
Payment	36.3 %	9.1 %	54.6 %
Order Status	16.7 %	25.0 %	58.3 %
Delivery	0.00 %	25.0 %	75.0 %
Stock Level	6.3 %	18.8 %	74.9 %

Table 3: Partition of code for the TPC-C benchmark.

Now, we proceed to do a sensitivity analysis assuming a hierarchical computing system built with available technologies. We assume a main processor running at 1 GHz, memory processors running between 100 and 500 MHz (*i.e.* $2 < \tau_2 < 10$), and disk processors running between 66 and 250 MHz (*i.e.* $4 < \tau_3 < 15$). Considering the database sizes of some implementations of transaction processing systems, we selected a configuration with four memory processors and 32 disks (which is expressed by $\Psi(1, 4, 32)$). The first section of Table 2 shows the maximum and minimum speedups obtained for different partitions of the code (*i.e.* varying values of Φ). Depending on the distribution of work, we can observe speedups of up to $4.52x$ when comparing with a traditional uniprocessor system. If the memory and disk processors are very slow, potential slowdowns may be observed for certain partitions of the code. For instance, the distribution $\Phi(15\%, 25\%, 60\%)$ encounters a slowdown when $\tau_2 = 10$ and $\tau_3 = 15$. However the same configuration is capable of a speedup of $2.86x$ given more powerful nodes.

Code partition is dependent on the nature of the transaction, and as such is relatively hard to change without restructuring the algorithm. It is then important to analyze the effect that changing the hardware configuration has over the speedup for a given code partition. This is shown in the lower half of Table 2. Here we work on a partition $\Phi(10\%, 20\%, 70\%)$ and change the number of memory and disk modules. Given the large amount of code given to the disk modules, we observe that incrementing the number of disks results in an increase in the speedup. As in any other multiprocessor system, the increase is not linear with the amount of processing elements added. It may also be noted that active memory systems and active disk systems are subsets of the proposed general hierarchical computing system.

6 Related Work

During the 1970s, computer scientists looked at database applications and proposed specially designed machines to handle the increasing gap in the performance between primary and secondary storage as well as the overwhelming software complexity present in database applications [28, 29]. Known as *Database Machines*, these systems incorporated specialized components in the form of processors per-disk, per-track, per-head and associative memories, in order to facilitate the access of data. The problem with these systems was that the use of non-commodity hardware drastically increased the cost of the systems. Additionally they were designed to handle only database workload, which resulted in a declined interest by the rest of the architecture and software communities.

Database machines saw their last days with the development of parallel databases [30], which proved to be a cost-effective solution for the problems of the day. Since then, modern commercial databases have adopted several of the proposed algorithms: parallel sort [31], parallel join [32] and other algorithms that tradeoff memory utilization for I/O bandwidth [33].

In addition to database machines, several research projects have looked at the idea of having computation elements close to the data. Intelligent memories have mostly been targeted to regular numeric applications [9, 10], but recent attempts also look at its use in non-regular applications [11, 34, 12, 35]. Likewise the idea of intelligent disk has also been covered by different research groups. The workloads considered for this technology consist of Decision Support Systems [13, 14, 15, 17], data-mining and multimedia applications [16].

Related to our research, the X-Tree architecture [36, 37], looks at a multiprocessor organization, where processors are connected using a binary tree. This topology facilitates the design of high bandwidth systems as the average distance between the nodes increases logarithmically with the number of nodes in the system. However, the emphasis in the X-Tree system was to build VLSI chips based on the idea of recursive architectures [38], where it was possible to design a computer system by constructing a hierarchy with the same type of processors. Another example of a recursive system is the Data Driven Machine (DDM1), designed by Davis et al.[39]. DDM1 was able to exploit concurrency due to its implementation of Data Driven Nets (DDN), which constitute a form of dataflow similar to the one used by our model.

While the computing paradigm presented in this paper has similarities to the aforementioned research efforts, it must be noted that the merits of several past architectural paradigms are being synergistically combined and applied to transaction processing in our current research effort. We are leveraging on advancements in active memories and active disks while at the same time taking advantage of the advancements done during the last twenty years in parallel databases, and query optimizers.

7 Conclusions

In this paper, we have presented a hierarchical model of computing, which is based on the concept of performing computations in a hierarchical manner distributed over the memory hierarchy. This research is intended to alleviate the imbalance of computation and data accessing experienced by large scale transaction processing systems. The building blocks that help to realize the proposed hierarchical model are active memory units and active disk devices that are emerging in the market. The basic principle is to use computation engines that sit close to the location of the data and the use of a hierarchy to connect these computation engines. This paradigm also brings in benefits of the dataflow model of computation.

We described the computation paradigm and outlined a simple code partitioning scheme. Then we applied the code partitioning scheme to the TPC-C benchmark, and observed that it is possible to split the transactions using static information about the database tables and storage capacity of the computation nodes. Using the simple partitioning algorithm and assuming inexpensive memory and disk processors, we show that a hierarchical computing system with four memory processors and 32 disk processors can obtain speedups of up to $4.52x$ when compared with traditional uniprocessor systems. While performance degradations will occur in non-parallelizable code, or systems with very slow memory or disk processors, it is seen that judicious partitioning of code can yield in performance improvements. Since transaction processing has been shown to contain significant amounts of parallelism, partitioning code in a fruitful way is not immensely difficult. In summary, hierarchical computing which exploits parallelism, distributes computations, and reduces the data transport requirements is a feasible model of computation for future database servers.

One attractive result of using this paradigm is the feasibility of using processors, with different performance ratings in the same system. This is possible due to the use of heterogenous processing elements in the different layers of the hierarchy. Processors which are used at the top of the hierarchy move down as memory processors once a new high performance processor generation arrives. Simultaneously, the current memory processors will move to the disk as disk processors. This maximizes the lifetime of a processor design, amortizing the cost of the design.

References

[1] R. Winter, "The growth of enterprise data: Implications for the storage infrastructure," *in Whitepaper Winter Corporation*, 1998.

[2] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," in *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA, USA), pp. 205–216, Oct. 2–7 1998.

[3] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski, "Contrasting characteristics and cache performance of technical and multi-user commercial workloads," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA, USA), pp. 145–156, Oct. 4–7 1994.

[4] S. E. Perl and R. L. Sites, "Studies of Windows NT Performance Using Dynamic Execution Trees," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, (Seattle, WA, USA), pp. 169–184, Oct. 28–31 1996.

[5] L. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, (Barcelona, Spain), pp. 3–14, June 27–July 1 1998.

[6] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta, "The impact of architectural trends on operating system performance," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, (Copper Mountain, CO), pp. 285–298, ACM Press, Dec. 1995.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?," in *Proceedings of the 25th Conference on Very Large Data Bases (VLDB'99)*, (Edinburgh, Scotland), pp. 15–26, Sept. 7–10 1999.

[8] K. Keeton, D. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, (Barcelona, Spain), pp. 15–26, June 27–July 1 1998.

[9] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A memory-SIMD hybrid and its application to DSP," in *Custom Integrated Circuits Conference*, pp. 30.6.1–30.6.4, May 1992.

[10] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM: IRAM," *IEEE MICRO*, Apr. 1997.

[11] M. Oskin, F. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, (Barcelona, Spain), pp. 192–203, June 27–July 1 1998.

[12] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Drapper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freech, J. Shin, and J. Park, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proceedings of the High Performance Networking and Computing Conference (SC99)*, (Portland, OR), Nov. 13–19 1999.

[13] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (IDISKs)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, (Seattle, WA, USA), pp. 42–52, June 1–4 1998.

[14] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA, USA), pp. 81–91, Oct. 2–7 1998.

[15] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA, USA), pp. 92–103, Oct. 2–7 1998.

[16] E. Riedel, G. Gibson, and C. Faloustos, "Active storage for large-scale data mining and multimedia," in *Proceedings of the 24th VLDB Conference*, Aug. 24–27 1998.

[17] G. Memik, M. T. Kandemir, and A. Choudhary, "Design and evaluation of smart disk architecture for DSS commercial workloads," in *Proceedings of the 2000 International Conference on Parallel Processing*, (Toronto, ON, Canada), pp. 335–342, Aug. 21–24 2000.

[18] Cirrus Logic, Inc., "Preliminary product bulletin CLSH8665." June 1998.

[19] Intel Corporation, "i960 HX microprocessor developers's manual." Order Number 272484-002, Sept. 1998.

[20] Siemens Microelectronics, "TriCore architecture overview handbook." Feb. 1999.

[21] A. Tessardo, "TMS320C27x: New generation of embedded processors looks like a microcontroller, runs like a DSP." White Paper SPRA446, Digital Signal Processing Solutions, 1998.

[22] E. F. Codd, "A relational model for shared large data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[23] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," in *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages* (J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, eds.), Berlin, DE: Springer-Verlag, 1987. Lecture Notes in Computer Science 259.

[24] "TPC-C specification."
http://www.tpc.org/cspec.html.

[25] "IBM DB2 Universal Database."
http://www.software.ibm.com/data/db2/udb/.

[26] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing Survey*, vol. 16, pp. 111–152, June 1984.

[27] J. Lee, Y. Solihin, and J. Torrellas, "Automatically mapping code on an intelligent memory architecture," in *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, (Monterrey, Mexico), Jan. 19–24 2001.

[28] D. K. Hsiao, ed., *Advanced Database Machine Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

[29] L. L. Miller, A. R. Hurson, and S. H. Pakzad, eds., *Parallel architectures for data/knowledge-based systems.* Los Alamitos, CA: IEEE Computer Society Press, 1995.

[30] D. J. DeWitt and J. Gray, "Parallel database systems: The future of high-performance database systems," *Communications of the ACM*, vol. 35, pp. 85–98, June 1992.

[31] M. H. Nodine and J. S. Vitter, "Greed sort optimal deterministic sorting on parallel disks," *ACM Transactions on Database Systems*, vol. 42, pp. 919–933, July 1995.

[32] A. Segev, "Optimization of join operations in horizontally partitioned database systems," *ACM Transactions on Database Systems*, vol. 11, pp. 48–80, Mar. 1986.

[33] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Transactions on Database Systems*, vol. 11, pp. 239–264, Sept. 1986.

[34] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: Toward an advanced intelligent memory system," in *Proceedings of the International Conference on Computer Design (ICCD99)*, (Austin, TX, USA), Oct. 1999.

[35] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, (Vancouver, BC, Canada), pp. 161–171, June 12–14 2000.

[36] A. M. Despain and D. A. Patterson, "X-TREE: A tree structured multi-processor computer architecture," in *Proceedings of the 5th Annual International Symposium on Computer Architecure*, (Palo Alto, CA, USA), pp. 144–151, Apr. 3–5 1978.

[37] D. A. Patterson, E. S. Fehr, and C. H. Sequin, "Design considerations for the VLSI processor of X-TREE," in *Proceedings of the 6th Annual International Symposium on Computer Architecure*, (Philadelphia, PA, USA), pp. 90–101, Apr. 23–25 1979.

[38] P. C. Treleaven, "VLSI processor architectures," *IEEE Computer*, pp. 33–45, June 1982.

[39] A. L. Davis, "The architecure and system method of DDM1: A recursively structured data driven machine," in *Proceedings of the 5th Annual International Symposium on Computer Architecure*, (Palo Alto, CA, USA), pp. 210–215, Apr. 3–5 1978.