The Dissertation Committee for Juan Carlos Rubio
certifies that this is the approved version of the following dissertation:

Exploring the Potential of a Hierarchical Computing Model for a Commercial Server

Committee:

Lizy Kurian John, Supervisor

Douglas C. Burger

Joydeep Ghosh

Kimberly Keeton

Ann Marie Maynard

Yale N. Patt

Exploring the Potential of a Hierarchical Computing Model for a Commercial Server

by

Juan Carlos Rubio, B.S.E., M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2004

To my parents and brothers,

for their great support

Acknowledgments

I am grateful to my advisor, Prof. Lizy John for her guidance and patience during this research. I also appreciate the interest she showed for the well-being of her students, and how she shared her perspective about research with all her students.

I thank Doug Burger, Joydeep Ghosh, Kimberly Keeton, Ann Marie Maynard and Yale Patt for being part of my dissertation committee. I feel honored for having a stellar committee and value all the comments they offered. These comments and some of their questions helped me present this dissertation in a clear and objective way.

I want to thank the current and past members of the Laboratory for Computer Architecture for their insight and support during my tenure as a graduate student. Conversations with them were a core component of my learning experience and contributed to more than one idea published during those years. Also, the comments they offered during several talks shaped the content and presentation of this dissertation. Thanks to Melanie, Shirley, Debi and Gem for helping with numerous administrative issues during my time in graduate school.

Thanks also to Tom Keller and Mootaz Elnozahy for their advice. I con-

sider myself very lucky to have met such talented people during my internships at the IBM Austin Research Lab. Presenting my work to them sharpened my presentation skills and helped me focus on the main issues of my research. Their industry perspective helped me understand the tradeoffs present in my research. I also need to thank other researchers from IBM ARL for their comments and help. Particularly Charles Lefurgy, for his collaboration with the data placement techniques, and the team that ported SimOS to the PowerPC platform.

A special mention goes to my parents, who taught me that exploring and thinking are two key components of the learning process. These qualities were extremely helpful while working on this research. During these years of graduate school, their love and support provided a safe harbor which made the experience more manageable.

This work was also possible thanks to Matt Smith, with whom I had many conversations about my research and life. Though Matt is not a computer scientist, his impeccable reasoning sparked more than one idea. Matt also served as my *full time counselor* during the stressful months before the dissertation, for what I am really grateful.

JUAN CARLOS RUBIO

The University of Texas at Austin
August 2004

Exploring the Potential of a Hierarchical Computing Model for a Commercial Server

Publication No. _____

Juan Carlos Rubio, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Lizy Kurian John

Computer servers are an important driving force in the computer industry. Virtually any major enterprise, such as airlines, banks, or product vendors, depends on servers for such core procedures as selling and distributing products or managing a workforce. Several hardware and software innovations have made their appearance in the context of computer server systems before they were adopted by desktop platforms. The server market has also driven the design of new system architectures. A large fraction of the computer server systems used today are running commercial workloads such as decision support systems (DSS), on-line transaction processing (OLTP) and web servers. Commercial workloads access large amounts of data, imposing heavy demands on the memory and storage sub-systems. As a result, there is a large amount of traffic in I/O and memory buses, which hurts the performance and scalability

of the system.

This dissertation investigates the data movement problem in a computer server system running a commercial workload. To reduce the amount of data transferred between the storage subsystem and the processors, processing units are distributed across the memory/storage hierarchy. A programming model is proposed to facilitate the decomposition of large tasks into simple operations. These operations are distributed through the layers of the hierarchy depending on the affinity of the operation to a particular layer. A task mapping heuristic is proposed for efficient mapping of operations into the various processors. They are executed by the assigned processors, and results are made available to the higher layers, where subsequent operations can be performed.

We evaluate the effectiveness of the proposed Hierarchical Computing model using the SimOS full system simulator. On a group of TPC-H like queries, Hierarchical Computing systems achieve speedups of up to $1.22x$ over comparable 8-processor CC-NUMA systems. We show that the improved execution is due to a reduction of data traffic over the global interconnects. The Hierarchical Computing model also shows good scalability for larger configurations. Comparing an HC system with 31 processors to a 32-processor CC-NUMA system shows speedups between $1.14x$ and $1.45x$.

This dissertation also presents a data placement optimization to be used together with the Hierarchical Computing model or in a conventional CC-NUMA multiprocessor system. This technique uses information about the tasks that run in the system and tries to obtain a good layout to reduce the amount of global data transfers.

The effectiveness of the data placement optimizations is evaluated using a CC-NUMA system with 16 processors, where we obtain speedups of 4 to 13%

over a stripped data layout. Likewise, a Hierarchical Computing system with similar processor, memory and storage resources shows speedups of 15 to 30% over the HC system with a non-optimized layout, and 23 to 56% over the CC-NUMA system with a non-optimized layout.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The server market is the driving factor for many technological advancements in the computer industry. This is true for the memory and disk sectors, and particularly true for the microprocessor sector. The server market also fosters the design of new system architectures, as in the case of symmetric multiprocessor systems, which were initially used for servers and have recently been adopted in high-end desktop systems.

A few years back, the server market was dominated mainly by systems running technical workloads. But during the last two decades, it has changed to power a large portion of commercial operations [83]. According to this study, commercial applications accounted for approximately 80% of the computer server market back in 1995. The same study expected an additional 15% annual growth in the high-end server market until the year 2000. Overall computer sales decreased after the "dot-com burst". But recently, the popularity of commercial applications has contributed to a new increase in servers sales worldwide [65, 50].

Commercial applications such as on-line transaction processing (OLTP)

and business decision support systems (DSS) are driving the development of powerful server systems. OLTP systems are used to handle tasks required during the routine functioning of a business (e.g., a client buys products; the managers check the inventory or adjust the price of an item). On the other hand, DSS systems are used to generate composite information 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).

Although both workloads fit within the category of transaction processing, they have many differences. While OLTP operations usually have datasets on the order of kilobytes or megabytes, DSS operations usually access megabytes or hundreds of megabytes of data. Recent literature suggests that DSS systems will be accessing terabytes in the near future [97]. OLTP operations are of short duration, taking milliseconds to complete, whereas DSS operations take minutes. The number of concurrent operations in an OLTP system is on 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, note the delivery of a package). DSS systems, on the other hand, use mostly read operations during their execution.

1.1 Transaction Processing Systems

Transaction processing systems are typically implemented using a multi-tier architecture, as shown in Figure 1.1. This figure also shows how clients on the left are connected to an intermediate or *middle-tier* server through a switched network. The function of the middle-tier server is to act as a filter and

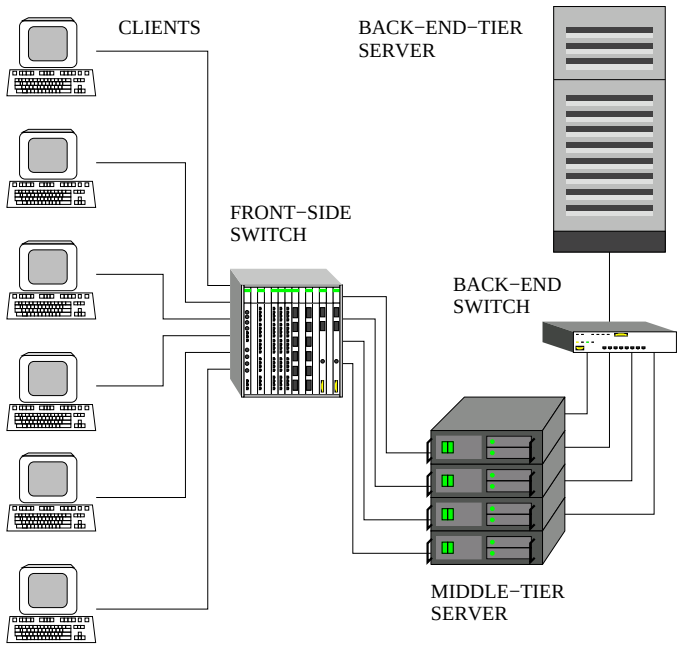

Figure 1.1: A conventional multi-tier system architecture used in a transaction processing system.

reject those requests presented by the clients that are incorrectly generated. It also enforces the security in the system and serves as a parser, transforming requests formulated in one language domain (e.g., HTML) to another (e.g., SQL).

The final component of the system is the *back-end-tier*, which is the focus of this research. 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.

1.2 Challenges for Commercial Server Design

In terms of their execution, commercial workloads are different from technical workloads and present more vigorous demands on the memory and storage sub-systems [57, 7]. In fact, research analyzing commercial workloads indicates that systems spend a significant fraction of the execution time waiting for I/O devices to access the data [32, 33]. Once the data is brought to main memory, the processor uses a substantial amount of the remaining execution time to handle memory accesses [4].

1.2.1 Amount of Data Transferred

One reason for this imbalance between computation and data access is entrenched in the principles of traditional memory hierarchies, where data moves from the storage sub-system to main memory, and from there to the CPU, 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 for a commercial workload.

Many modern servers are shared memory multiprocessors (SMP) [19, 35], or clusters of SMP nodes [72, 27, 16]. Commercial applications running on these platforms transfer a large amount of data from storage to the CPU over the global interconnect. We observe that caches cannot hold the entire dataset, and queries that require multiple passes often end up retrieving the same data multiple times. Figure 1.2 illustrates this for a group of TPC-H like queries [92]. These queries will be discussed in detail in Chapter 5, but one can observe the enormous amount of data transferred, which can be more than 4 times the dataset size.

Figure 1.3 shows more details about the data transferred in the 2x4

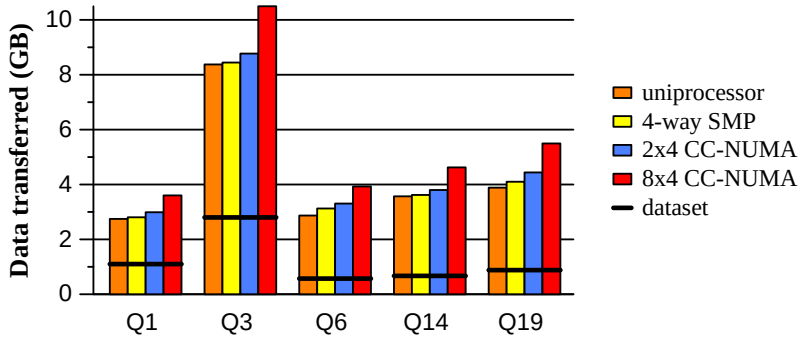

Figure 1.2: Amount of data transferred between the processors and memory while executing a group of DSS queries. The dataset line marks the size of the dataset. The workload and configuration of the systems are described in Chapter 5.

CC-NUMA configuration. The *dataset* component represents the first time data is accessed. Since accesses are done at the granularity of a page, *page padding* indicates the additional bytes appended to the desired data. *Capacity* represents the accesses done by the processor to data that has already been in the caches. Software writes *temporary* data back to memory (and frequently to disk), in anticipation that it will be used again, but not immediately. Finally, *coherence* is the data used by the multiprocessor system to maintain a consistent view of memory. As this figure shows, a large amount of the data

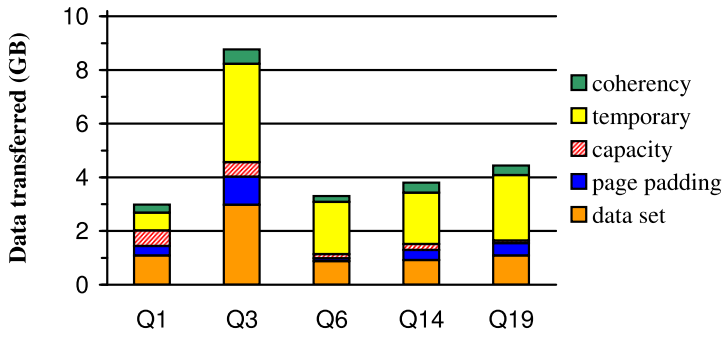

Figure 1.3: Classification of the data transferred between the processors and memory in the 2x4 CC-NUMA system

is transferred because the memory/storage hierarchy cannot hold all the data used by the workload (*capacity* and *temporary* results).

1.2.2 Interconnect Contentions

Also, depending on the configuration of the interconnect, a significant number of cycles is spent by the processors in bus contention. This is particularly important for most enterprise servers. Figure 1.4 shows that the average number of processors waiting for the bus increases dramatically with system size. Moving data between storage and computing elements creates a bottleneck. Clusters improve scalability, but do not solve the basic problems of data transport.

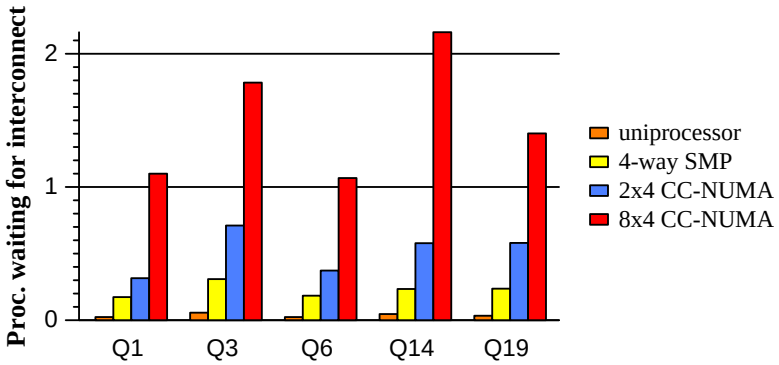

Figure 1.4: Average number of processors waiting for the bus. The workload and configuration of the systems are described in Chapter 5.

1.3 Hierarchical Computing

This dissertation presents the Hierarchical Computing model (HC) as a solution to the problems presented above. The HC model promotes the use of local data whenever possible. To accomplish this, computing is distributed

across a computer system's memory/storage hierarchy. Processors are integrated together with the disk and memory controllers. Then, as a server handles a task, operations are distributed across these processors, which can operate on the data close to them. Their results can then be used by other processors to complete the assigned task.

1.4 Thesis Statement

Placing processors across the memory/storage hierarchy can reduce the amount of data transferred over the global interconnect. This reduces contentions and significantly improves system performance.

1.5 Contributions

The Hierarchical Computing model presented in this dissertation makes several contributions in the form of a hardware and software framework that can be used to run a commercial workload. The following list summarizes the contributions:

- **Hierarchical Computing model**
  - We conducted an analysis of the different phases of a DSS workload. The key observation is that a well-tuned computer system running a DSS workload transfers between 2 to 4 times the required data. Furthermore, most of this data has been already accessed during the life of the query.
  - This dissertation presents the design of a computer system that places processors across the memory/storage hierarchy. It describes the communication mechanisms between the processors in the system, including the handling of signals and management of buffers.

- We also propose an execution model for this new system architecture. This model is based on a decomposition of a task into a group of simpler operations. The operations are then assigned to different processors across the system. The dissertation also includes guidelines to provide a programming model suited for the workload.
- The potential of the HC system for a decision support system workload is then evaluated using full system simulation. The DSS workload is based on a group of queries from the TPC-H benchmark, which runs on top of IBM DB2.

- **Task mapping**
  - To quickly map the operations of a complex task (e.g.,database query) over the HC system, we develop a task mapping heuristic. This heuristic shows good results for the studied decision support system workload.
  - A methodology is proposed to easily obtain a group of empirical time equations based on the properties of the data being accessed and the operations performed over it. These time equations allow the task mapping heuristic to estimate the total time of performing a task in an HC system.

- **Data Placement Optimization**
  - We demonstrate that a simulated annealing technique [46, 1] may be used to arrive at a good data layout based on information about the workload that will run in the system. This heuristic can be applied to HC and conventional CC-NUMA systems.

1.6 Organization

This dissertation is organized as follows. Chapter 2 presents a summary of prior research that is related to this work. Chapter 3 presents the Hierarchical Computing model, explaining the operation of the hardware, communication of the devices and programming model. Chapter 4 presents a technique that allows us to map code to the processing elements in a Hierarchical Computing system. Chapters 5 describes the experiments used to evaluate this idea and then results are presented in Chapter 6. Chapter 7 presents an automatic data placement optimization based on the simulated annealing method. This optimization is applied on both conventional and Hierarchical Computing systems. Chapter 8 concludes the dissertation by highlighting significant results and areas for possible future research.

Chapter 2

Related Work

The following sections present prior literature that is closely related to this research. This chapter has been split into four sections: System Architectures, Mapping of Operations, Data Placement and Analysis of Commercial Workloads.

2.1 System Architectures

The Hierarchical Computing system we investigate in this dissertation uses a hardware-software approach that allows it to reduce the amount of data transferred globally by the system. This section presents architectures that address the same problem or that result in similar hardware implementations.

2.1.1 Database Machines

During the 1970s, computer scientists studying database applications proposed specially designed machines to handle the increasing gap in the performance between primary and secondary storage [31, 63]. Known as *Database Machines*, these systems incorporated specialized components (e.g., processors per-disk, per-track, and per-head, and associative memories) in order to

facilitate the access of data. These components allowed the system to efficiently access data from secondary storage or perform common operations over this data (e.g.. join, sort). Unfortunately, their high degree of specialization sharply limited interest among the larger architecture and software communities. In addition, the use of non-commodity hardware made database machines prohibitively expensive, and the systems never came into widespread use.

DIRECT [21, 22] is a multiprocessor database machine designed around a MIMD architecture. The use of multiple instruction streams allows it to exploit inter-query concurrency, while the use of multiple data streams permits it to exploit intra-query concurrency. The machine is designed around a *matrix interconnect*, which permits any of the processors (PDP/11) to access a group of CCD modules [1]. In a way, the architecture of DIRECT is similar to that of a large scale SMP system, except that permanent storage and coherence were handled by the host processor.

Recursive machines were popular as a way to simplify the design of a computer system. In this approach, a simple flexible module is designed and used extensively throughout the system. One example, the DDM1 machine [18], is especially relevant to this work. It used data driven nets as its machine language, which is similar to the HC programming model described in Section 3.2. These nets could represent large amounts of concurrency and pipelining, as the modules in the multiprocessor system were completely asynchronous.

[1] Charge-Coupled Device (CCD): MOS device made arranged in such a way that the electric charge output of one cell charges an adjacent one. They were used as pseudo-associative memory during 1970-1980. Today they are used as light sensors in digital photography.

2.1.2 Tree Organizations

The X-tree machine [20, 71] developed at UC Berkeley is one of the key examples of Tree machines of the late 70s. This machine is another recursive architecture, formed by a group of modular components called *X-nodes*. Each X-node has a processor, some memory and a routing interconnect. The X-nodes are structured as a balanced binary tree, where communication between the nodes is done using messages.

Harris, et al. [30] also explore the design of a computer system that follows a tree organization. Each node has a processor and some amount of memory (DRAM). This organization is selected as it allows an extra order of magnitude in the number of processors before communication problems affect the system. Implementing interrupts (except for resetting the nodes) was seen as detrimental to the performance of the system. So in contrast with the X-tree machine, the communication between the nodes is controlled directly by the processors. They explore the use of this system to find solutions to the traveling salesperson problem and checker move selection. One of the criticisms of tree architectures during this period was that the interconnect at the top was not capable of providing enough bandwidth to the processors. However, the authors found that the processors at the root are not used as often as the leaf processors, reducing the impact of the bandwidth problem.

The aforementioned tree machines never got into mainstream computer design, and current server designers usually do not think of a tree architecture as a suitable topology for a commercial server.

2.1.3 Intelligent Devices

Intelligent memories have targeted regular numeric applications [53, 23, 68], and recent projects are starting to look at their use in non-regular applications [66, 40, 28, 55]. Also several research groups have focused on the use of intelligent disks [43, 2, 74, 59].

Dynamic Associative Access Memory (DAAM) [51, 53] puts small processing elements (not a processor) near the sense amplifiers of a DRAM chip. The main use of these chips is to massively search data stored in them. The initial design of a 1 Mbit DRAM (organized as 256x4096) expands the 10 transistors of the sense amplifiers to 24. Their calculations indicate that a large array could access a 1 TB database and search it in 60 μs. Later work [52] reveals that the search operations could be done simultaneously with the refresh cycle of the memory cells.

Computational RAM (C-RAM) [23] adds bit-serial SIMD processors to the sense amplifiers of an otherwise conventional DRAM chip. This is designed to exploit the internal bandwidth of the memory chip. As the study suggests, the memory bandwidth at the sense amplifiers is up to 3 orders of magnitude higher than that at the processor level. The authors also fabricated an 8 Kbit prototype and showed the performance for regular applications (DSP).

The EXECUBE architecture [47, 48, 85] is based on a processor-in-memory (PIM) chip built using a 0.8 μm trench cell CMOS DRAM technology. The chip has 4 Mbit DRAM and 100 K logic gates, organized as 8 processing elements (PE) connected as a 3-D hypercube (connections to 4 of its neighbors). Each PE has one 25 MHz CPU, two 32 Kx9 DRAM macros and an additional external link, which permits the chips to be connected as a massively parallel system without any glue logic. The PEs can operate as

a MIMD computer (fetching instructions from its own memory), or in SIMD mode (when using an external broadcast bus).

The Intelligent RAM (IRAM) project [68, 69, 70] is probably the best known effort to combine processors and memory. This approach integrates a single processor inside a memory chip built using DRAM fabrication technology. The objective is to increase the available bandwidth and reduce the latency to memory. IRAM's first implementation is a vector processor, which shows the benefit of this idea for regular applications.

The goal of Active Pages is to off-load data manipulation to logic in the memory subsystem [66]. The authors propose an execution model that augments a memory page with a set of functions to operate on that data. Their first conceptualization of the idea was RADram (Reconfigurable Architecture DRAM), which includes reconfigurable logic (as an FPGA[2]). They expect to use an MDL process (Merged DRAM logic), which would result in a $10x$ slowdown of the logic with respect to the main processor. The second instantiation of Active Pages [67] proposes to replace the FPGA with simple processors by a simplified scalar MIPS R3000 or a VLIW processor. The scalar processor has the lowest power consumption of the three, while the FPGA results in greater speedups. They conclude that the VLIW implementation offers the best of both worlds.

FlexRAM [40, 87] tries to give applications greater flexibility by providing one central processor per chip (P.mem) and several simple processors next to the sense amplifiers (P.array). A P.array can see its memory and the memory of its two neighbors. P.mem can move data within the chip to facilitate certain operations. The system also has an inter-chip network, which allows a

[2]Field Programmable Gate Array

P.mem to access any memory location in the system. Their experiments indicate that a system with four FlexRAM chips can run between 4 to 25 times faster than a system with conventional DRAM.

Riedel, et al. [74] mention that current disk drives have very efficient processors in the disk units. They propose a system called Active Disks, which uses processors at the disk units to run application-level code. Their experiments use separate slower computers to emulate Active Disks. The server is a regular computer with directly attached SCSI drives. All computers are connected using an Ethernet switch. Their evaluation for data mining, multimedia and scan operations shows speedups that scale linearly with the number of Active Disks used in the system.

The IDISK project [43] from UC Berkeley proposes to use an additional processor at the disk unit to perform operations on behalf of the main application. The disk units are connected using point-to-point serial interconnects, resulting in a flat cluster of disks. This resembles a computer cluster, except the clustering is performed at the disk level rather than at the computer level. Keeton et al. [42] indicates that using an IDISK system could be beneficial for a DSS workload.

There is another Active Disk project [2] based at the University of Maryland. The goal of this project is to move certain computations to the disk in order to off-load the host processor. Under this model, the host processor is used to coordinate the disks, schedule operations to them and combine the results. The authors propose a stream-based programming model that uploads user-defined *disklets* to the processors on disk. A disklet operates on a stream initiated on its behalf by the host processor. To guarantee the reliability and security of the system, certain safeguards are imposed on the disklets (i.e., they

cannot allocate memory or initiate I/O). Disklets can skip portions of a stream by interacting with the layer of the operating system that runs in the disk. Experiments on a group of basic database operations show good scalability for systems with up to 32 Active Disk modules. They also show that the communication between the disks is crucial for operations that combine data (join, sort and cube) [95].

2.1.4 Clusters

For scalable applications, *clusters* [72] are a popular architecture. Most clusters that exist now have flat interconnection networks. At best, the interconnection network is hierarchical (e.g., the hierarchical switch in the Compaq AlphaServer GS320 system [27, 16]). This organization increases the bandwidth of the system.

Memik, et al. [59] evaluated the performance of smart disk clusters against traditional clusters. Their results indicate that smart disk clusters outperform traditional cluster architectures in most queries.

2.1.5 Heterogeneous Multiprocessor Systems

Researchers have explored heterogeneous multiprocessor systems [60] so an application can select the resources that best fit its needs. The Hierarchical Processors and Memory (HPAM) [8, 9] and the Heterogeneous Distributed Shared-Memory (HDSM) [25] projects from Purdue address the design of these systems. Both projects use traditional parallel applications such as SPLASH-2 [98]. The HPAM project explores the design of a computer system with a small number of fast processing resources and a large number of relatively inexpensive processors. The system is hierarchical in a conceptual sense, as processors may be viewed as comprising a hierarchy according to speed/cost.

This work was followed by the HDSM project, where the nodes of an otherwise conventional cc-NUMA system are populated with different numbers and types of processors and memories. It uses a conventional interconnect, so any processor can access any memory location regardless of the node.

2.2 Mapping of Operations

Lee et al. [49] proposes an automatic code partitioning heuristic that is applied to a a FlexRAM system. The authors analyze the loop structure of programs from the SPEC CPU benchmark and classify them according to the affinity of the loops to either the host processor or the FlexRAM modules. The affinity is estimated using simple cost functions based on the number of instructions and iterations of the loops, and the frequency of the processors.

Manegold et al. [56] presents models to estimate the cost of performing database operations in a system with a general hierarchical memory subsystem. They analyze the memory access patterns of the queries and develop generic cost functions for each pattern. Their model uses information about the latency and bandwidth of the memory components combined with the requirements of the access pattern to estimate the cost of the query.

Rădulescu et al. [75] presents the Critical Path Reduction heuristic (CPR). CPR is used to schedule an application over a distributed system to exploit its task and data parallelism. They identify *M-tasks* and *S-tasks*; the former being a task that can run in multiple processors, while the latter runs in a single processor. The iterative heuristic then identifies the critical path and allocates additional processors to the M-task that will most likely shorten it.

2.3 Data Layout

Combinatorial optimizations, including simulated annealing, have been used in the context of VLSI and Electronic Design Automation to solve placement and route problems [78, 79]. In the field of computer architecture, the uses of combinatorial optimizations have been limited mostly to genetic algorithms used to explore the design space of microprocessor components [82, 24, 5]. Swanson et al. recently conjectured that SA could be used to assist instruction scheduling and placement [86]. Simulated annealing has also been used to cluster data when performing pattern recognition [39], and to produce optimized query plans [38].

Work by Tsangaris et al. looked at stochastic techniques for clustering objects [93]. Additional work shows that randomizing algorithms produce the best clustering, but the cost involved is too high for their application [94].

A study by McErlean et al. shows that simulated annealing can also be used to cluster data in a database [58]. Their work used real runs in a database instead of a cost function. Therefore it required many hours to generate a single layout. Our work shows that picking an adequate cost function can produce a layout in less than a second.

Work by Rao et al. uses genetic algorithms to find a good partition for a group of queries in a shared-nothing database cluster [73]. A partition is a description of the logical group of nodes (nodegroup) over which tables and indices are evenly allocated. Our approach is similar to theirs if we think of a single node of our CC-NUMA system as a nodegroup. However, we opt for a finer grained approach that allows small chunks of data to be allocated unevenly across the different disks of the system instead of evenly splitting tables across each nodegroup. Additionally, their work considers partitioning

as a part of query optimization. We look at data reorganization as a way to optimize the layout further for an existing set of query plans.

2.4 Analysis of Commercial Workloads

A significant amount of research has been done to identify the characteristics of commercial workloads. Research by Maynard, et al. [57] was among the first to appear in the literature. They used an RS/6000 system with an in-order processor to evaluate the cache behavior of an OLTP workload (TPC-A [88], TPC-C [90], and Laddis [81], among others). These results were compared with those of a group of proprietary technical applications and two benchmarks from the SPEC92 suite [80]. They found that the thread switching activity associated with commercial applications tend to diminish the temporal locality of instructions streams. Also, different threads tend to require different data, which applies additional pressure over the cache hierarchy.

Barroso, et al. [7] analyzed the memory performance of SMP systems running OLTP (TPC-B [89]), DSS (TPC-D [91]), and web index (AltaVista) workloads. They found that the processors running the OLTP workload results in a very large CPI (7.0). This CPI is larger than that of commercial systems running the real TPC-C benchmark. They also observed that increasing the size of the Bcache (board cache – equivalent to an L3 cache shared by all processors) helps the OLTP workload as it reduces the latency of instruction fetches. The DSS workload, on the other hand, has a lower CPI (1.5 to 1.9), and behaves well with the existing Scache (second level cache). The web index behaves similarly to the DSS workload. They also claim that "operating system activity and I/O latencies do not dominate the behavior of well-tuned

database workloads". We believe this statement is inaccurate as their system was designed to be memory resident [3]. Another problem with this study is their claim that TPC-B has a similar behavior to TPC-C. Later, the same group of researchers compares the behavior of the TPC-B and TPC-C benchmarks in their experimental infrastructure [84]. This work acknowledges that the CPI of a TPC-B workload is higher. It also indicated that the time spent in kernel functions can be high as 20%.

Keeton, et al. [41, 44] analyzed the performance of a 4-way SMP system running the TPC-C benchmark. They showed that the operating system accounts for 20% of the execution time. The overall average CPI of the benchmark was 3.39, which was skewed by the 6.48 cycles of the OS-CPI. They also indicate that the branch predictor in the processor does not work as efficiently as with the SPEC CPU benchmark. This might be due to the high context-switch rates and use of non-looping branches of the workload. Likewise, the workload does not appear to benefit from the the out-of-order components of the processor, showing a majority of cycles where no instructions are decoded or retired. The authors suggested that a narrower issue width might be sufficient for the workload. It would be interesting to investigate the use of processors capable of holding more in-flight instructions (e.g., Pentium 4). Finally, they observe that a modest bus utilization (over 60%) can hamper the memory subsystem by increasing the average memory access latency (from 97 cycles to 111 cycles in a 4-way SMP system). In that situation, increasing the size of the L2 caches reduces the bus utilization, thus increasing the scalability of the system.

Ailamaki, et al. [4] analyzes the behavior of fundamental database op-

[3]The largest database was 900 MB and they used a system with 2 GB of memory.

erations using four commercial database management systems (DBMS). They use simpler select and join operations, which are customized to resemble OLTP and DSS workloads. The workload is strictly memory resident. Their work disagrees with previous work, as they mention that the L2 cache has a bigger impact on data accesses, and instruction fetches are affected by the L1 cache. This discrepancy is caused by the design of the experiments, as the L1 cache is only 16 KB, and the workload only considers a single OLTP operation at a time.

Cao, et al. [12] studies a 4-way SMP system running the TPC-D benchmark. They observe that the average CPI (1.27) is comparable to that of SPEC CPU applications. They found that branch prediction works very well for this workload, in contrast with an OLTP workload. And similar to Ailamaki et al., they found that the small L1 cache constitutes a bottleneck for the instruction fetching [4]. They also show that different queries of the benchmark can have very large L2 cache misses (35.2%), and that the percentage of L2 misses is related to the cycles the processor is stalled. However, they do not show a relationship between these stalls and the CPI of the queries.

Hankins, et al. [29] presents a methodology to scale an OLTP workload so the behavior of the processor resembles that of a deployed system. They observe that an OLTP workload can operate in 3 regions: CPU bound (with a few warehouses, which results in a small dataset), balanced, and I/O bound. They study the number of instructions per transaction (IPX) and the CPI of the system as they scale the number of warehouses (size of the database). By monitoring those 2 factors, it is possible to designate a *pivot point*, where CPI

[4] Ailamaki, et al. [4] uses a Pentium II Xeon with a 16 KB L1 instruction cache, whereas Cao, et al. [12] uses a Pentium Pro with an 8 KB L1 instruction cache.

and IPX cease to increase linearly. Their main observation is that if the scaled system is larger than the pivoting point, the database is representative of a full OLTP system. Other results agree with previous work.

Chapter 3

Hierarchical Computing

We present the Hierarchical Computing (HC) model as a solution to the data transfer problem present in conventional systems. The HC model distributes processing elements across the memory/storage hierarchy. To take advantage of the parallelism present in the tasks that run in the server, the system decomposes tasks into simpler operations. These operations are then distributed and executed by the different layers of the hierarchy depending on the affinity of the task to a particular layer. By performing some operations closer to where data resides, an HC system reduces the amount of data transferred over the global interconnect. This can effectively reduce contention problems.

This chapter presents the system architecture, execution and programming model of an HC system. The next chapter describes the technique used to map the operations of a task across the processors.

3.1 System Architecture

A traditional system requires all processing to be performed at the top of the memory/storage hierarchy. As explained in Chapter 1, Figure 1.2

showed that commercial servers transfer large amounts of data over the system interconnect, while Figure 1.4 showed that a large number of processor cycles are lost because of contentions in the interconnect. It is the goal of the HC model to encourage the use of local accesses whenever possible by distributing computations across the memory/storage hierarchy. Processors are located in memory and disk, close to the location of the data. They are provided with a local bus to operate on the data, and communicate among themselves using a hierarchical interconnect (i.e., a tree). This interconnect matches the initial topology of the memory/storage hierarchy, which reduces the amount of additional communication links in the system.

Figure 3.1 shows the topology of a hierarchical computing system based on a binary tree interconnect. In this figure and across the rest of the dissertation, the term *storage* is used to indicate any device that can contain data; it applies both to permanent storage (e.g., disks, flash RAM, magnetic RAM) and to volatile storage (e.g., DRAM, SRAM). A *node* is formed by coupling the computing element (a single processor in this example) to the memory/storage module, or by using an integrated module like the ones mentioned in Section 2.1.3. We define a *layer* as the group of nodes that sit at the same logical distance from the top of the hierarchy. If all the nodes in a layer have the same hardware characteristics, we refer to the layer as *homogeneous*. A *symmetric* topology consists entirely of homogeneous layers. Through the rest of the dissertation we will be referring to a symmetric topology. The notation $Layer_L$ is used to represent a given layer L, where $Layer_1$ is the topmost layer. To identify individual nodes within a layer we use the notation $Layer_L\{N\}$, where N is the list of nodes.

Data accesses within the node are considered *local data accesses* and

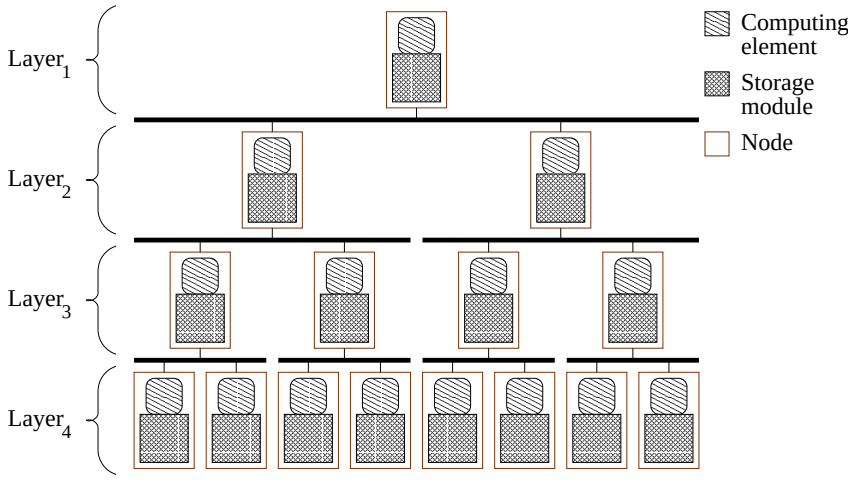

Figure 3.1: A generalized view of a hierarchical computing system (HC-1-2-4-8). For simplicity, the diagram shows each node with only one computing element, one storage module, and either 2 children nodes or none, but other configurations are also possible. We also present the interconnection between a parent node and its children nodes as a bus, but it could also be a more complex network.

usually result in a significantly lower access latency, as they bypass the global interconnect circuitry. Local accesses also provide a higher bandwidth as it is usually easier to provide high speed or wider buses for local modules [68].

An HC system can use commodity processing and storage components, which could be placed in the same package. The intelligence required in the different levels of the hierarchy could also be realized using intelligent memory modules investigated in recent research [53, 23, 68, 66, 40, 28, 55] and intelligent or active disks [43, 2, 74, 59]. Computing capability can be realized in the network or switches using network processors [37, 64], micro-controllers, or similar chips embedded in the switch/bus interface.

3.2 Execution Model

For presentation purposes, this section describes the execution model in terms of a decision support workload and the database software that runs in a server system. This execution model could be applied to other workloads and platforms as well.

The database server runs in one or more of the processors that sit at the top layer of the hierarchy. Once the database server receives a request to execute a query, it prepares an *execution plan*. This execution plan describes the operations that will be applied to the tables of the database to execute the query. The first step is the *decomposition* of the query into simpler operations [17, 10]. Then the order of execution of the operations is decided (*query scheduling*). This includes verifying that accesses to a table currently being modified are deferred until the completion of pending operations. Once the operations are scheduled, they are ready to be executed. It is at this point that an HC system differs from a conventional system. Whereas a conventional system will execute each one of the operations in the main processors, an HC system will run some of them in the processors across the storage hierarchy. At this point, the HC system takes the query schedule provided by the database software and maps each of the operations to a particular processor in the system. Chapter 4 presents the details of this operation.

Figure 3.2 shows three layers of a sample HC system. Data is partitioned to allow a high level of distributed computation. For illustration purposes, we will assume that all requested data resides in the intermediate layer ($Layer_2\{1, 2\}$ of Figure 3.2). A processor in the top layer ($Layer_1\{1\}$) starts the distribution of the operation by allocating a response buffer. This buffer is used to hold any data generated by the lower level. After the buffer

is set, it issues a command (CMD) to one or more nodes in the intermediate layer ($Layer_2$). Commands include enough information to allow nodes of the intermediate layers to perform the sub-queries. A command is really an index to one of the data handling routines that have been previously loaded into that layer by the database. After the command is sent, the node is free to operate on other tasks while it waits for data.

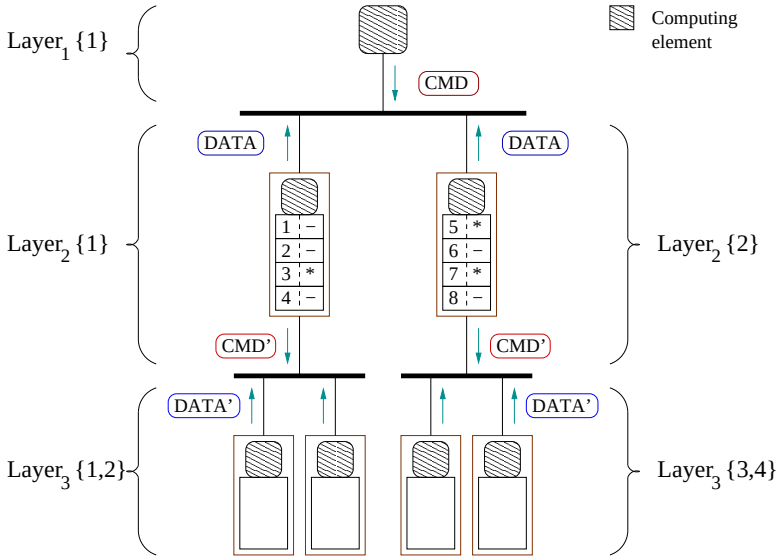

Figure 3.2: Execution of an operation in the Hierarchical Computing model.

Once a node receives a command from a higher layer, it checks the data in its storage module. If the data is not present within local storage, the process repeats downward. In effect, it performs a preorder traversal starting at the top of the hierarchy. The node prepares a command and sends it to the next layer, forwarding all responses to the higher layer. When a leaf node does not contain the requested data, a null response is sent.

To facilitate the communication of the data, the hardware provides basic

control flow signals. These help the top layer handle the amount of data that might result from a delegated operation.

All commands are tagged by the originating node with a unique identifier. The responding node also tags the response based on the tag of the initial command and the ID of the node. This technique is similar to the tokens used in traditional dataflow machines [6] and active messages [96]. The system can initiate operations in different nodes, allowing multiple independent operations to be performed in parallel. A node can receive multiple commands, and can execute them when the required data is available. Because there is no requirement that results return in order, they are distinguished by their tags. This mode of operation allows for a pipelined out-of-order execution similar to the one seen in modern microprocessors. To reduce the overhead of processing responses, we also tag a command with the *ID* of the layer initiating it.

Finally, the model implements a namespace locator in the form of a *software-managed table allocation index*. This index permits a processor to quickly locate data within its local storage. It is also used to determine if data is not present, thus avoiding a lengthy traversal of the data.

Minimizing the movement of data is accomplished by performing computations in the processing element closest to the data whenever possible. However, some operations may be performed at a different layer. This occurs when an operation benefits significantly from a more powerful computing element (e.g., a high frequency dynamically scheduled processor) or requires resources not available in the computing element (e.g., floating point units), or for combining partial answers supplied by other nodes. The idea that different operations benefit from different types of resources has been explored for traditional parallel processing workloads such as SPLASH [60, 8, 9, 25]. We

further discuss this and other related work in Section 2.1. Chapter 4 discusses in more details the issue of mapping the code in a Hierarchical Computing system.

As described earlier, the database manager loads ahead of time a set of data handling routines that will be used by the different nodes across the hierarchy. These routines are selected according to their use in modern transaction processing workloads. The current interface provides an implementation of the most common operations:

- Selection: locates records within a single table that match a particular criterion
- Join: merges the results of two or more selections or tables.
- Sort query results by some criterion
- Insert, remove, and update records from an existing table

To handle the different SQL operations and data manipulation algorithms, we use two types of operation primitives: *individual* and *aggregate*. These operations are based on the execution model presented in this section, and differ in the way the results are generated by the lower node and interpreted by the upper.

- Individual Primitive

  During the execution of the operation, the lower layer informs the upper layer of every single result. It effectively acts as an unbuffered non-combining filter. An example is searching a range of data for a string. The semantics allow the operation to return on the first event triggered

or to continue operating until it reaches the end of the region. Figure 3.3 shows the flow of commands and results for this primitive.

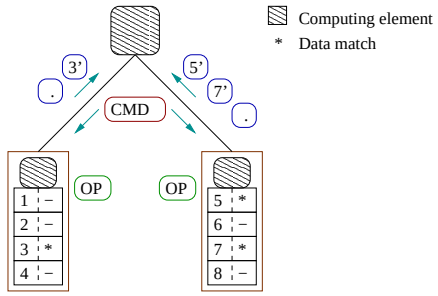

Figure 3.3: Individual Primitive.

- Aggregate Primitive

  For this primitive, the lower layer accesses its associated data and finds those elements matching a particular criteria. However, it does not send all these results to the upper layer. Instead it sends an aggregate result once all the data has been analyzed. In the context of parallel processing, this operation is also known as *data reduction*. The most common aggregate functions in commercial workloads are: *sum*, *count*, *average*, *max*, and *min*. The *average* function is a special case, as it returns a pair of values, *sum* and *count*. The upper layer computes *average* from all returned *sum* and *count* values. Figure 3.4 shows the flow of commands and results for the aggregate primitive.

3.3 Addressing and Coherence

To manipulate the data in the system, we use a global virtual address space. That is, all processors share the same mapping of virtual to physical addresses. However, certain processors might be able to access only physical

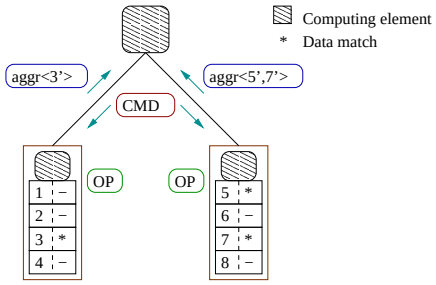

Figure 3.4: Aggregate Primitive.

addresses that are local to their node. We say that the processors in the node *own* that range of addresses. As we will soon explain, this policy facilitates the maintenance of coherence in the system.

To support this model, the operating system (OS) uses a global set of address translation tables. For explanation purposes we continue presenting the case of processors in memory, but this can be applied to processors in disk as well. The OS must initialize the memory-processor modules before they can be used. The sequence of steps required to initialize a processor in memory is as follows:

(i) The OS allocates a continuous region of memory on the range owned by that device.

(ii) Next it prepares an address translation table for the portions of the virtual address space that physically reside in the memory local to the device.

(iii) It then pins down those pages so they are not evicted from memory.

(iv) The memory processor is instructed to use this address translation table.

(v) A semaphore is used to indicate that the global address space should not be modified for that range

If a change to the address translation tables is required, the OS must guarantee that the memory-processor modules affected are informed of the change. Currently, this process requires the OS to monitor the state of the address range semaphore. This is performed as follows:

(i) The OS access the semaphore for that range

(ii) If the semaphore is not set, the OS updates the corresponding tables.

(iii) If the semaphore is set, it verifies the processors that own that range are not halted.

(iv) If the processors are not halted, it waits for them to complete their work before updating the tables.

(v) If the processors are halted, it resets the processors, update the corresponding tables and re-initialize the processors.

In this system, coherence is maintained using a hardware/software approach. The basic premise is that processors must not share data simultaneously if at least one is expected to modify it. Additionally, before allowing a processor to use the data, the system must guarantee that the data is current. If the routine running in one processor accesses data, and the system instructs another processor to regain write access of that data, the system flushes the caches to make the copy on memory current. While the second processor accesses the memory pages in write mode no other processors are allowed to read from it.

3.4 Security

Commercial applications require privilege control for the data they store. In the HC system, this is allowed by having meta data that is kept in the same storage module as the data. This meta data contains information that assists the database in determining the ownership of a group of rows. Commands are then sent with an identifier that indicates the originator of the query. The code that runs in the data location is then responsible for checking the privileges of the originator over the data. To guarantee that this ownership is respected, the data handling routines are provided by the database server.

Chapter 4

Task Mapping

4.1 Introduction

The previous chapter presented the Hierarchical Computing (HC) architecture and its execution model. In an HC system, the operations that comprise a task (in this discussion, a database query) are executed in processors across the memory hierarchy. Some processors are closer to the data, which makes them prime candidates to execute an operation that uses that particular data. Additionally, processors might have different computation or storage capabilities, which can also affect the performance of the operation. The goal of the task mapping module in a HC system is to determine in which nodes operations should be executed.

This problem is closely related to a multiprocessor task scheduler [62, 15, 99, 77]. For almost all situations, the scheduling problem has been shown to be NP-complete [11]. But in most situations, it is possible to analyze the system and obtain an approximated solution (i.e., a schedule for which its cost approximates the cost of the lowest possible cost) [76]. For a Hierarchical

Computing system, the task is particularly challenging as the nodes may have different characteristics regarding CPU performance, data capacity, communication channels and data available. It is then necessary to determine an adequate mechanism to map the operations onto the system. A heuristic is used to obtain a good solution. The goal of this heuristic is to execute the operation in the processor that executes it most efficiently without hurting the performance of the whole task.

The workload used in this research consists of a group of queries executed by a database manager. Commonly, queries are represented using a high level language like SQL. It is the job of the database manager to parse the query and break it into simpler operations – process known as *query decomposition*. The next step is *query scheduling*, where the order of the operations is decided. The output of this stage is the *query execution plan* (QEP), which constitutes the input for the heuristic.

4.2 Heuristic

A common scenario for an HC system involves the simultaneous processing of operations from multiple tasks (e.g., queries). At some point, an incoming task is presented to the system in the form of a *task plan* (e.g., query execution plan). The *task mapper* is then responsible for planning a good mapping for this task plan. To arrive at this mapping, it uses information about the incoming task, static information about the system, and dynamic information about active and pending operations. Figure 4.1 shows this task mapping mechanism as a feedback system. The *task scheduler* provides information about the current operations and the scheduled operations. The *enable* gate confirms that the task mapping is permissible and introduces it to the

task queue.

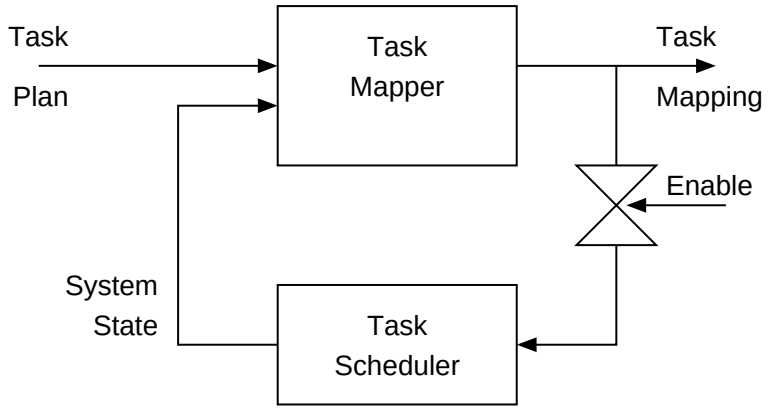

Figure 4.1: Representation of the task mapping module as a feedback system.

A divide and conquer heuristic is used to map operations across the nodes of the system. The input to the heuristic is a *task*. The task is represented as a directed graph of N operations and their requirements. Figure 4.2 shows the representation of a sample task. The system also uses an architectural description of a system to define the parameters of the mapping module. This description is specified as a list of M nodes and their properties. The requirements for an operation are specified as properties of the data that would be accessed and a characteristic function. Figure 4.3 shows the description of a sample system. For the workload we are studying, the properties of the data include number of rows the operation accesses and the width of the row. The characteristic function is an expression used to estimate the time required to perform the operation in a particular node of the system. The details of this function are presented in Section 4.3.2.

The goal of this heuristic is to reduce the total execution time of the task. To accomplish this, it tries to reduce the execution time for the critical

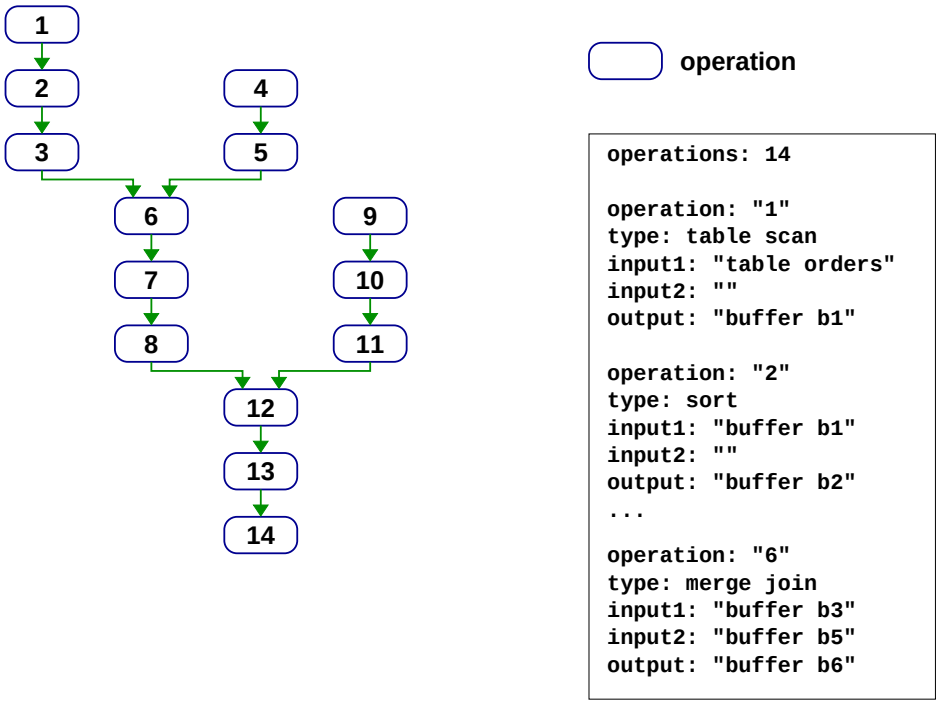

Figure 4.2: A sample task for the task mapping heuristic. (a) Representation of a task as directed graph of operations. (b) Formal description of the operations in a task.

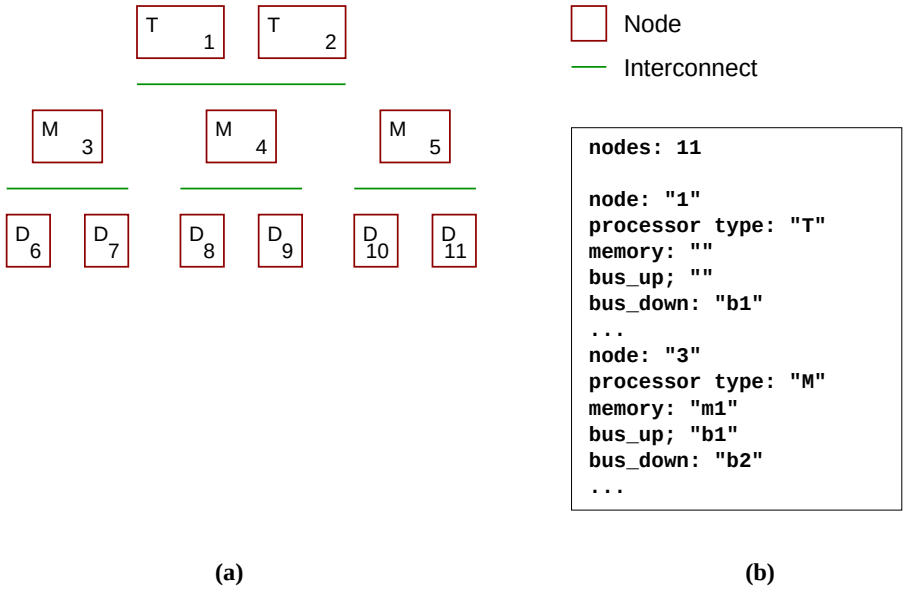

Figure 4.3: A sample system for the task mapping heuristic: (a) graphical representation, and (b) formal description.

path without drastically affecting the latency of the other paths. Figure 4.4 shows the basic algorithm used to map the task across the nodes of the system.

Figure 4.5 shows a graphical representation of a map that reduces the time for the sample task (Figure 4.2) on the sample HC system (Figure 4.3). The task map is divided in 3 horizontal regions; one for each level of the hierarchy of the system. The width of the region depends on the number of processors on the layer. Operations are represented by bubbles and numbered according to the query execution plan. Operations shown using a dotted line belong to previously scheduled tasks, thus preventing the current operations from using the corresponding resources. Time is represented along the x-axis, with a dotted line to represent the present time. Operations to the left of the line have already completed.

(i) For each operation, estimate cost of running in every node

(ii) Divide the task graph into straight segments. Estimate minimum and median cost of each segment

(iii) Starting at ready nodes, propagate median cost of segment

- When reaching intersections, select the segment with greatest cost to determine critical path

(iv) Map operations in the critical path to the processor resulting in the earliest completion

(v) Remove critical path from graph and repeat (from 3) for remaining operations

Figure 4.4: Task mapping algorithm.

4.3 Cost functions

This section explains the process used to obtain the cost functions for the operations that constitute a task. Our approach uses an operational database system where sample operations are performed. The results from the experiment are then used to obtain empirical expressions for the time in terms of parameters of the workload.

4.3.1 Tasks and Operations

The task mapping module in a Hierarchical Computing system receives the query plan as input. The individual operations that form the query plan are the operations considered by the task mapping module. Table 4.1 shows the operations considered by this task mapping heuristic. These operations are the most common components found in the queries of a DSS workload.

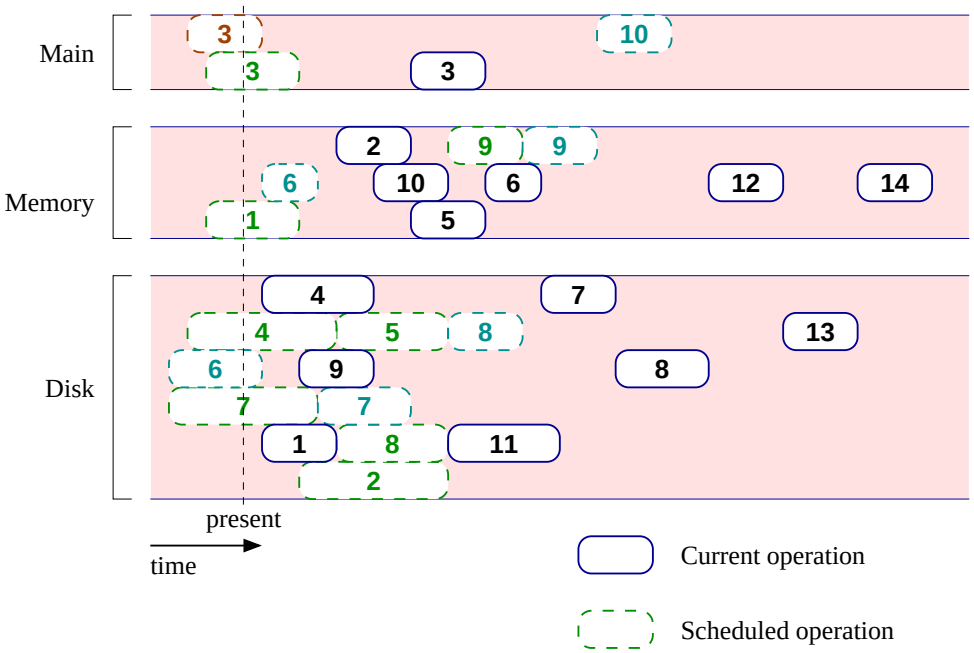

Figure 4.5: Representation of a sample task map. The dotted bubbles are the operations currently running in the system, the regular bubbles are the spots that would be assigned to the new task.

Table 4.1: Operations studied for the task mapping heuristic.

Operations	Implementation
table scan	predicate: 1 integer predicate: 1 string predicate: 1 date predicate: 2 integer
index scan	predicate: 1 integer predicate: 1 string predicate: 2 integer
sort	quick bubble
unique	hash
join	merge join nested loop join hash join

4.3.2 Characteristic Function

To apply the task mapping heuristic, we need to estimate the cost of performing an operation in a given processor of the system. This estimation could be done using an analytical model, where a real operation is analyzed and a cost function is then generated. This process would require us to analyze each function and consider the impact that the configuration of the nodes might have on that operation. But this method is error prone and could potentially result in large error margins.

Instead, we use an empirical analysis of the functions. In this analysis we run a series of operations in a real system and measure their execution time. We then run a series of experiments with the intention of parameterizing the resulting time as a function of: properties of the operation, characteristics of the architecture, and location of the data. These experiments test the following

parameters:

- **Processor configurations** - This parameter considers the different types of processors in the HC system. Those explored in this dissertation are main processor (1 GHz), memory processor (500 MHz) and disk processor (250 MHz). In this experiment, the processors have identical cache configurations and differ only in their operating frequencies.

- **Datasets** - This parameter controls table sizes by adjusting the number of rows in the tables. These experiments produce a group of cost functions that are independent of the number of rows. This is explained in more detail below.

- **Operations** - This parameter determines the algorithm used to access the data. The algorithms considered in this work are listed in Table 4.1.

The resulting time function is not a linear combination of the above stated parameters. The presence of caches, TLBs and finite amounts of memory results in a non-linear behavior of the time function as the dataset is changed. To reduce the impact of this variation, I set up five ranges of table sizes determined by the layers of the memory hierarchy. A linear dependence of time on the dataset size can be expected within each range. That is, a single linear expression can be found to represent the time needed to operate on tables that fit within a given range. The full range is not used to avoid hitting boundary conditions. The characteristics of the five ranges are described in Table 4.2. Measurements are performed using three table size within each range: upper limit, lower limit, and the geometric mean of the range.

The three processor configurations correspond to those listed in Table 5.3. Since performing this analysis requires a large number of experiments,

Table 4.2: Characteristics of the database tables for the cost function measurements. The average width of the rows is 100 bytes. The memory page size is set to 8 KB.

Description	Number of rows
Fit in L1 cache	50 - 300
Fit in L2 cache	700 - 2600
Fit in TLB	3500 - 5200
Fit in memory	5500 - 6000
Does not fit in memory	8000 - 12000

I chose to use a real computer system instead of software simulations. This approach causes a larger error for the estimated cost function, but is sufficient for the purposes of this study. The configuration of the testing system is listed in Table 4.3. The system has a single processor that runs at 1 GHz, and has an option in the BIOS that can slow it down to 500 MHz. To simulate the 250 MHz processor, a process is set to execute an infinite loop and execute at a 50% duty cycle, thus reducing the efficiency of the processor.

Table 4.3: Configuration of the system used to estimate the cost functions for the task mapping heuristic.

Processor	1 GHz 3-way issue 64 KB L1-D cache
Memory	256 MB, 100 ms 2 banks 64-bit, 200 MHz bus
Storage	2 disks, 16 GB 10,000 RPM 64-bit, 66 MHz PCI bus 1 Ultra160 SCSI controller

The 13 operations used are listed in Table 4.1. The data resulting from

the experiments is then run through a linear regression analysis statistical package to obtain a group of parameterized expressions. For the *table scan*, *index scan*, *sort* and *unique* operations, the expression has the form:

$$t = t_{row\ independent} + t_{row\ dependent} R \tag{4.1}$$

where R is variable that represents the number of rows in the table. $t_{row\ independent}$ is a parameter obtained from the linear regression, which does not depend on the number of rows. $t_{row\ dependent}$ is the second parameter or coefficient obtained from the regression. For the *join* operations, the expression is:

$$t = t_{row\ independent} + t_{row\ dependent\ outer\ table} R_1 + t_{row\ dependent\ inner\ table} R_2 \tag{4.2}$$

Here R_1 is the number of rows in the *outer table* (normally the largest). Likewise, R_2 represents the number of rows of the *inner table* (or elements in the hash structure for the hash join).

This analysis is performed for each processor configuration and memory range. Based on the memory ranges shown in Table 4.2, a boolean vector **S** is used to indicate the current memory range. Using this memory ranges allows us to decouple the analysis from a fixed table size. And using a boolean vector simplifies the notation of the expressions. The elements of the vector can be either 0 or 1 depending on the conditional statement shown in Equation 4.3.

$$S = \left| \begin{array}{rlll} size(L1D) > & size(dataset) & & ?\ 1:0 \\ size(L2) > & size(dataset) > & size(L1D) & ?\ 1:0 \\ size(TLB) > & size(dataset) > & size(L2) & ?\ 1:0 \\ size(mem) > & size(dataset) > & size(TLB) & ?\ 1:0 \\ & size(dataset) > & size(mem) & ?\ 1:0 \end{array} \right| \tag{4.3}$$

The coefficients obtained in these experiments are tabulated in Appendix A. The expression used to estimate the cost of operations that access a single table is given by Equation 4.4.

$$t(op, R) = \left| \begin{array}{l} t_{op,1,1} + t_{op,2,1}R \\ t_{op,1,2} + t_{op,2,2}R \\ t_{op,1,3} + t_{op,2,3}R \\ t_{op,1,4} + t_{op,2,4}R \\ t_{op,1,5} + t_{op,2,5}R \end{array} \right| \mathbf{S} \tag{4.4}$$

Likewise, Equation 4.5 shows the expression used for the join operations.

$$t(op, R_1, R_2) = \left| \begin{array}{l} t_{op,1,1} + t_{op,2,1}R_1 + t_{op,3,1}R_2 \\ t_{op,1,2} + t_{op,2,2}R_1 + t_{op,3,2}R_2 \\ t_{op,1,3} + t_{op,2,3}R_1 + t_{op,3,3}R_2 \\ t_{op,1,4} + t_{op,2,4}R_1 + t_{op,3,4}R_2 \\ t_{op,1,5} + t_{op,2,5}R_1 + t_{op,3,5}R_2 \end{array} \right| \mathbf{S} \tag{4.5}$$

Appendix A shows the values for these parameters. If the data is not reported as present for that accessible by that node, the reported time is ∞.

Whenever possible, operations are mapped using a floating map, which allows the system to re-map them after new tasks enter the system. To allow for this, a map consists of the *soft starting time* and a *hard starting time*. The soft starting time corresponds to the earliest time that an operation can be mapped. The hard starting time is the latest time that an operation can be mapped without affecting the remaining operations in its task map.

Chapter 5

Evaluation Methodology

The experiments used in this dissertation are based on performance simulation of systems running a decision support system (DSS) workload. Our simulator is based on the port of the full system simulator SimOS to the PowerPC ISA [36]. The simulator has been extended to model flat and hierarchical interconnects. The database management system used is IBM DB2 [34] version 6.1. The experiments consist of PowerPC based systems running AIX 4.3.

This chapter includes a description of the configurations of conventional cache coherent non-uniform memory access server systems (CC-NUMA), that are used as baseline configurations, and Hierarchical Computing systems. It describes the configuration of the database software and the workload used as well as the process used to adapt them to the HC configurations.

5.1 Baseline System Configurations

We are interested in comparing our work with multiprocessor systems, such as those used in state of the art servers to run commercial workloads. The experiments model server systems that are commonly used to run this

workload. To put these results into perspective, we include a uniprocessor system as well.

5.1.1 Organization

The following configurations were used to represent conventional server systems.

- **uniprocessor**, a system based on the configuration shown in Table 5.1.
- **4-way SMP**: a symmetric multiprocessor system with 4 processors. The memory is shared and on the other side of the bus. All other characteristics are similar to the uniprocessor configuration, as explained in Table 5.1.
- **2x4 CC-NUMA**: a CC-NUMA system of two 4-way SMP nodes. The resources of the system are similar to those of the previous configurations. They are split among 2 nodes, so each node has 4 processors, 512 MB of memory and 7 disks. The nodes are then connected with a high speed interconnect that incorporates a directory to maintain cache coherence. Figure 5.1 shows a diagram of this configuration. The latency across the network is modeled as 50 ns [61]. And Table 5.2 shows the resulting latencies for an unloaded system. Our initial experiments showed that a high number of remote instruction fetches reduced the performance of this configuration. So in this configuration we use larger instruction caches of 256 KB compared to the caches of 128 KB used in the previous configurations.
- **8x4 CC-NUMA**: a CC-NUMA system of eight 4-way SMP nodes. This configuration is a scaled version of the previous configuration (2x4 CC-

NUMA). It has eight nodes, which are identical to the nodes of the 2x4 CC-NUMA configuration. That gives the system a total of 32 processors, 4 GB of memory and 56 disks. The latency across the network is modeled as 100 ns [61]. Table 5.2 shows the latencies for an unloaded system.

Table 5.1: Configuration of the base systems.

	uniprocessor	4-way SMP	2x4 CC-NUMA	8x4 CC-NUMA
Processors	1	4	8	32
Clock rate	1 GHz			
Execution	out-of-order			
Issue width	4			
Function units	4 int, 4 fp			
L1-I	128 KB, 64 B lines, 2-way		256 KB, 64 B lines, 2-way	
L1-D	128 KB, 64 B lines, 2-way			
L2	4 MB, 128 B lines, 4-way			
Memory	1 GB			4 GB
	100 ns, 4 banks		100 ns, 4 banks each node	
	4 KB pages			
System bus	128 bits, 200 MHz, pipelined, split transaction			
I/O bus	64 bits, 66 MHz, PCI			
Disk I/O controller	2			8
Disk bus	160 MB/s, Ultra160 SCSI			
Disk units	9.1 GB, 3 ms latency			
system	2			8
database	8			32
logs	4			16

The interconnect used in the CC-NUMA configurations is capable of sustaining 720 MB/s per link per direction.

Table 5.2: Communication latencies for the CC-NUMA configurations.

Parameter	Latency	
	2x4 CC-NUMA	8x4 CC-NUMA
Local L2	35 ns	35 ns
Local memory	70 ns	70 ns
Local cache-to-cache	85 ns	85 ns
Remote memory	220 ns	270 ns
Remote cache-to-cache	235 ns	285 ns

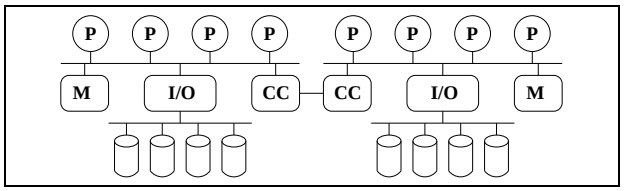

Figure 5.1: Diagram of multiprocessor configuration (2x4 CC-NUMA). The coherence controller (CC) contains a network interface controller (NIC), which is used to connect both 4-way SMP nodes.

5.1.2 Software

These experiments use IBM DB2 [34] version 6.1, which is a high performance commercial database product. The database server is configured to exploit the maximum parallelism available in each configuration. The tables are stored in an OS-managed tablespace, which is spread across all the disk units assigned to data in Table 5.1. So, in the first 3 configurations, the database tables and indices spread across 8 disks. The large configuration (8x4 CC-NUMA) has 56 disks, which are divided equally among all the nodes. Of the 56 disks, 32 disks are used to hold the tables and indices. The buffer pool [1] for the database is sized to approximately 70% of the available physical

[1]Buffer pool: a region of memory used by the database manager software to cache tables or indices, which are stored normally on disk.

memory. This allows the database to perform disk prefetching, which remove artificial I/O bottlenecks. The database manager system runs as a user process and the database manager system creates four working threads per CPU, which communicates using shared memory.

5.2 Hierarchical Computing System Configurations

To study the benefits of the Hierarchical Computing model, we select configurations with similar computation and storage resources as the baseline configurations introduced in Section 5.1. The software is also selected to require as few changes as possible.

5.2.1 Organization

The HC configurations are named according to the number of processors in each layer (e.g., an HC-1-2-4 configuration has 1 main processor, 2 memory-processor modules and 4 disk-processor modules). They are designed with the same amount of storage and computation resources as their counterparts whenever possible. However, in some cases we have opted to use slightly less memory (to use memory modules available on the market) or fewer processors (to avoid uneven division of data among the disk nodes). All choices were made conservatively for the HC systems; in no case does an HC configuration benefit from greater memory or processing capacity than the corresponding CC-NUMA configuration. The following HC configurations are used:

- **HC-1-3-0** is an HC system with 1 main processor and 3 memory-processor nodes. Each memory-processor node has 320 MB, which can be organized as a 256 MB DIMM and a 64 MB DIMM. This setup results in a total amount of 960 MB, or 7% less memory than the base configuration. Figure 5.2 shows a diagram of this configuration.

- **HC-1-1-6** is an HC system with 1 main processor, 1 memory-processor node and 6 disk-processor nodes. It has the same amount of memory and number of disks as the 2x4 CC-NUMA configuration. However, we only have processors in 6 of the 8 disks used to store the database. The memory-processor node has access to 1 GB of memory. Figure 5.2 shows the diagram for this configuration.

- **HC-1-2-4** configuration has 1 main processor, 2 memory-processor nodes and 4 disk-processor nodes. Each memory-processor module has 512 MB of memory. We keep the same number of disks, and two processors are placed on each of the disk I/O controllers. Figure 5.2 shows the diagram for this configuration.

- **HC-1-6-0** is an HC configuration with 1 main processor and 6 memory-processor nodes. Each memory-processor node has 160 MB, which can be organized as a 128 MB DIMM and a 32 MB DIMM. This setup results in a total amount of 960 MB, or 7% less memory than the base configuration.

- **HC-1-0-6** is an HC configuration with 6 disk-processor nodes. As in the HC-1-1-6 configuration, only 6 of the 8 disks that hold the database are disk-processor nodes. This configuration and HC-1-6-0 are selected to study the impact of processor placement in an HC system.

- **HC-1-5-25** has 1 main processor, 5 memory-processor nodes and 25 disk-processor nodes. It has computation and storage resources similar to the 8x4 CC-NUMA configuration. In order to use more standard memory modules, each memory-processor node has 768 MB for a total

of 3.75 GB.

- **HC-1-6-24** has 1 main processor, 6 memory-processor nodes and 24 disk-processor nodes. As in the HC-1-5-25, each node has 768 MB for a total of 3.75 GB.

- **2x4 CC-NUMA-HCsw**, the 2x4 CC-NUMA system using a software implementation of the HC model (HCsw). The purpose of this configuration is to test the pairing of the HC programming model with a conventional system. Figure 5.3 shows a diagram of the system. The software implementation is explained below.

Processors in the memory and disk modules are modeled as being simpler and slower than the main processors. There are reasons behind this decision, the most important being:

- Area constraints. The size of a memory module is limited and adding an advanced microprocessor might not be possible without affecting the timing, performance, and cost of the memory module. High performance processors usually require cooling devices (e.g., heat sinks, fans), which can complicate their placement in a disk unit.

- Power dissipation. High-end power processors consume large amounts of power, which in turns heat up the devices. Memory modules fabricated using DRAM technology are particularly susceptible to variations in their operating temperature. Disk units have a nominal power budget, which can be jeopardized by adding a power hungry processor.

- Economics. The trend is to add characteristics to differentiate products from one another, thus adding a processor can be beneficial to a product

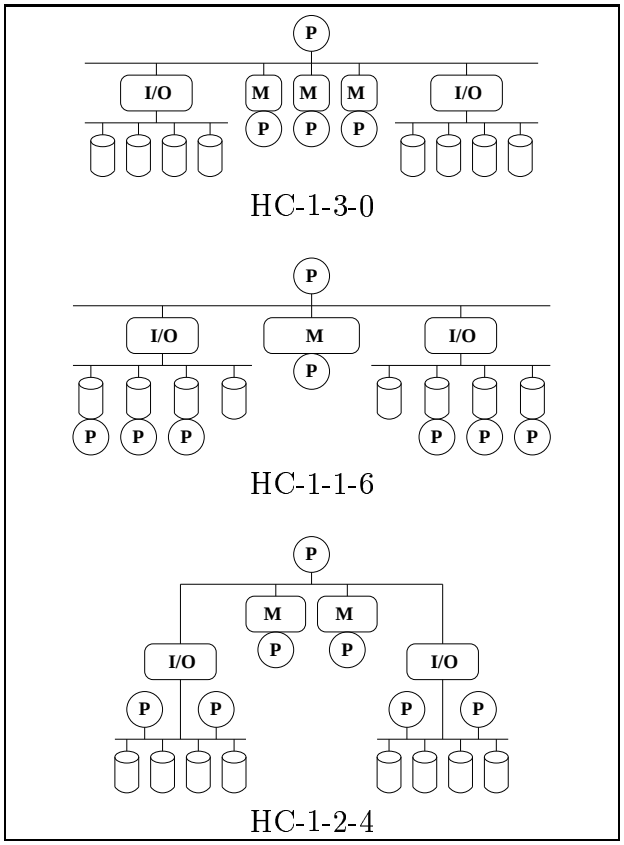

Figure 5.2: Hierarchical computing configurations (HC-1-3-0, HC-1-1-6, and HC-1-2-4).

line. However, using an expensive processor in a commodity device can increase the cost dramatically. Older cores are usually available at rates that would not drive up a product's price this way.

Table 5.3 shows the configuration for the processors used in the HC configurations.

5.2.2 Software

As in the conventional systems introduced in Section 5.1.2, the HC configurations use IBM DB2 as the database management system (DBMS).

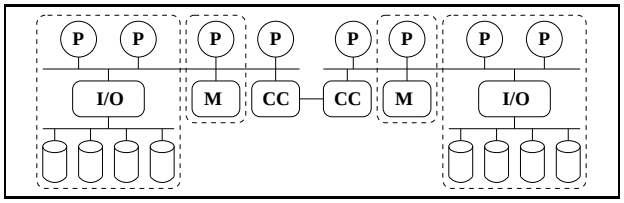

Figure 5.3: Distribution of tasks for the HC software on a CC-NUMA system (2x4 CC-NUMA-HCsw).

Table 5.3: Configuration of the processors in the HC configurations.

	main processor	memory processor	disk processor
Clock rate	1 GHz	500 MHz	250 MHz
Execution	out-of-order	out-of-order	in-order
Issue width	4	2	2
Function units	4 int, 4 fp	2 int, 2 fp	2 int, 1 fp
L1-I	128 KB, 64 B lines, 2-way		
L1-D	128 KB, 64 B lines, 2-way		
L2	4 MB, 128 B lines, 4-way		

This DBMS runs as a user process. However, we have modified the operation of the database to operate like an HC system. Since we did not have access to the database source code, we profiled the code to obtain the entry and exit points of key routine of the database. This information allowed us to control the node where a given routine had to be executed.

The software profile was done using the software profiler (*sprof*), which is part of the SimOS-PPC simulation infrastructure [36]. The following is a list of the profiled functions[2]:

- Query schedule hand-off: This function triggered once the query schedule is completed.

[2]The names we show do not correspond to the names selected by the developers of IBM DB2. Instead they were selected to describe the purpose of the routine. In some cases the same functionality was performed by more than one routine.

- Initiate data access: A functions that starts accessing the data in a table. The data can be in disk or memory.

- Load data from disk: this function is called by *initiate data access* if the data is not found in the buffer pool. It locates the disk that contains the data and starts the process of loading it to the buffer pool. This access is done asynchronously using direct memory access (DMA) transfers.

- Process data from memory buffer: A group of functions that are invoked over the data in the buffer pool to execute the operations comprising the query.

- Thread control (create thread, wake up thread, sleep, kill thread, fork): These functions manage the query processing and data access threads in the database. Their functions are similar to that of POSIX threads.

- Buffer pool management (set up, extend, reclaim): These functions are usually accessed when the DBMS initializes the buffer pools or undergoes tunning. Their behavior is similar to dynamic memory allocation in modern systems.

Once the addresses of the functions were determined, we set up a simulation to start the database and our control application. The function of this application is to keep processors busy with an infinite loop, which prevented the operating system from scheduling additional work to those processors without our consent. This application was given maximum priority, which prevented the OS scheduler from switching its context.

At the same time, the simulator tracks down the instruction addresses in all the processors. When the simulator reaches one of the profiled functions, it

makes a decision to continue executing the code there, free one reserved processor so the database will spawn a thread there, or remap memory. The decision was performed outside of the simulator using the TCL interface provided by SimOS-PPC. The four actions configured in the TCL script for the HC configurations are: buffer pool setup, query scheduler hand-off, initiate data access and end data access. Each actions involve several steps as described below:

1) **Monitor setup buffer pool**

- Determine address assigned by the database for buffer pool
- Determine a mapping so the buffer will stay local to the processor that will use it
- Use the control application to pin down the pages of that buffer in memory

2) **Monitor query scheduler hand-off**

- Use control application to obtain query plan used by the database
- Use control application to compute a task map
- Pass the task map to the TCL script

3) **Monitor start of initiate data access**

- Free processor that would execute the operation
- Hand-off execution to that processor

4) **Monitor end of process data**

- Reclaim processor

- Redirect processing to previous processor

These experiments use the task mapping described in Chapter 4. The resulting task maps usually place simple select operations at the disk level, a more complex selection at the memory level, and joins either by the top processor or memory processors, depending on availability.

5.3 Workload

To evaluate the ideas in this dissertation, we use a *TPC-H like* system [3]. Table 5.4 shows the characteristics of the database tables used by the benchmark with a scaling factor of 5. Conventional TPC-H databases use scalings of 1, 10, 100, and 1000. This database corresponds to approximately 5.7 GB of data. This scaling results in a database that is not memory resident. The cardinality column indicates the number of records in a database table. The next column shows the average size of a record in our implementation using the IBM DB2 database. This takes into account that variable length string fields are not padded with zeroes. The last column shows the size of the tables used.

We use a group of 5 TPC-H like queries, which are described in Table 5.5. To assess the strength of the HC model for modern server systems, we choose queries representative of conventional decision support scenarios. They range from relatively simple select queries (Q1 and Q6) to complex join operations (Q14 and Q19). To understand the architectural impact of the workload on the configurations we study, we examine the execution of each query. The

[3]The benchmark has been implemented according to the TPC-H specifications [92], and includes most optimizations used in similar commercial systems. However, it cannot be labeled as TPC-H as it has not been officially audited by the Transaction Processing Council.

Table 5.4: Dimensions of the tables for our implementation of a TPC-H like workload in DB2.

Table	Cardinality (records)	Average Row size (bytes)	Table Size (MB)
Nation	25	185	< 0.01
Region	5	181	< 0.01
Part	1,000,000	156	148.8
Supplier	50,000	168	8.1
PastSupp	4,000,000	163	625.0
Customer	750,000	186	133.2
Orders	7,500,000	121	868.1
LineItem	30,000,000	142	4076.1

operations of a query, and the dependencies between them, are commonly referred to as the *query execution plan* (QEP). Figures 5.4 through 5.8 illustrate QEPs for these queries.

Query Q1 is highly parallelizable and has a data set size of approximately 5.5 GB, which includes the base table and one temporary table created during the process. Q1 begins with a stage in which all the CPUs read the table from disk and perform range comparisons. This operation has little spatial locality, and requires a modest amount of arithmetic manipulations, which results in a high rate of data requests for a short period of time. The second phase of Q1 is the creation of a temporary table which is then sorted. This phase is also parallelizable, and since the temporary table has just been created, most of its records are resident in main memory. In addition, since the sorting process continuously operates on a group of elements [3], there is a perceptible amount of cache-to-cache data transfer in the shared memory configurations.

Table 5.5: Description of selected TPC-H like queries.

Query	Name	Data set size	Implementation
Q1	Pricing Summary Report	5.5 GB	A sequential scan of table *LineItem*. It generates a large number of aggregate values.
Q3	Shipping Priority	14 GB	A merge join of tables *Customer* and *Order* and a subsequent merge join of the result with table *LineItem*.
Q6	Forecasting Revenue Change	2.9 GB	An indexed scan of table *LineItem*.
Q14	Promotion Effect	3.4 GB	An indexed scan of table *LineItem* and a subsequent merge join with table *Part*.
Q19	Discounted Revenue	4.4 GB	A merge join of tables *Part* and *LineItem*.

Query Q3 performs two join operations using two of the largest tables of the benchmark; this results in a sub-linear speedup as the number of computing elements increases. Of the queries we use, this has the largest working set (more than 14 GB). This includes three base tables and four temporary tables, one of which is larger than the amount of physical memory in the system. This query is executed in five distinct phases, where the last two phases are responsible for more than 60% of the total execution time in any of the configurations. These two phases exhibit a poor memory and cache behavior which results in a large amount of bus traffic.

Query Q6 has a working set of 2.9 GB. This query uses an index to assist in finding requested records. At 225 MB, the index fits partially in main memory together with the table. But due to the high efficiency of the indexing scheme, processors request data at a faster rate than other queries,

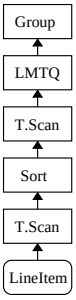

Figure 5.4: Execution plan for query 1 of the TPC-H benchmark.

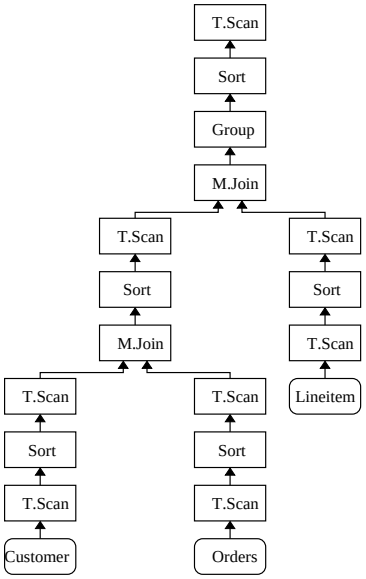

Figure 5.5: Execution plan for query 3 of the TPC-H benchmark.

which causes bus contention problems.

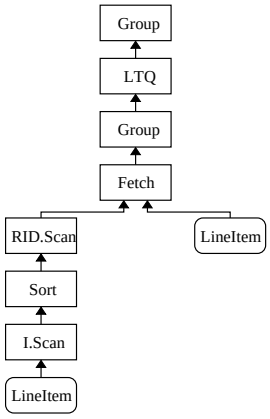

Figure 5.6: Execution plan for query 6 of the TPC-H benchmark.

Query Q14 has a working set of 3.3 GB, which includes data from one base index, two base tables and three temporary tables. Similar to Q6, this query uses an index in finding requested records; however, this phase constitutes a small fraction of the execution of the query. As in Q3, a large portion of the execution time (up to 50% in the uniprocessor configuration) is spent performing the join operation. Though the join operates on relatively small tables, they do not fit in a single L2 cache; however, they do fit comfortably in the combined L2 caches of the multiprocessor configurations. As a result this query can benefit from cache-to-cache transfers between the processors in the CC-NUMA configurations.

Query Q19 has a working set of 4.4 GB, which includes two base tables and two smaller temporary tables. In this query, the criterion applied to identify relevant records is very complicated. Although the amount of data required is no less than in the other queries, the consumption rate of the data is lower. This results in a continuous use of the bus, but without much

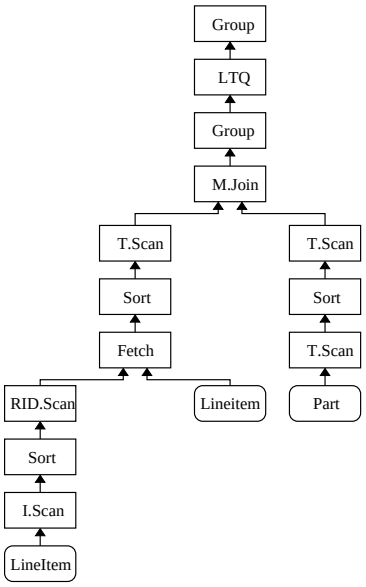

Figure 5.7: Execution plan for query 14 of the TPC-H benchmark.

contention among the processors.

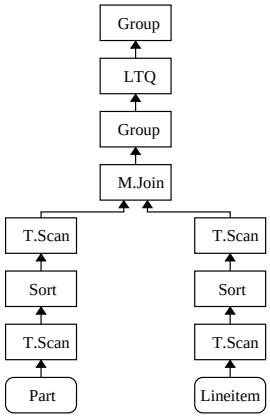

Figure 5.8: Execution plan for query 19 of the TPC-H benchmark.

Chapter 6

Results

This chapter presents the performance results for a group of Hierarchical Computing configurations. It analyzes these results and explains the factors that affect performance. It also presents the effectiveness of the task mapping heuristic and compares it with a perfect mapping.

6.1 Hierarchical Computing

This section analyzes the performance of the various configurations on the selected TPC-H like queries. We study the amount of data transferred and the resulting interconnect contention as factors affecting the performance of the systems. A conventional configuration (2x4 CC-NUMA) and a group of HC configurations (HC-1-0-6, HC-1-6-0, HC-1-1-6 and HC-1-2-4) are used to study the performance of HC systems on different kinds of queries. We conclude with a look at the scalability of larger systems (8x4 CC-NUMA, HC-1-5-25, and HC-1-6-24) and smaller systems (4-way SMP, HC-1-3-0, HC-1-1-2, and HC-1-0-3).

6.2 Performance of HC systems

We study the execution of the selected queries in configuration 2x4 CC-NUMA, which is found in many commercial enterprises. This configuration has 8 processors. We compare this to four HC configurations of similar size. Configuration HC-1-0-6 was picked to exemplify the advantages of the HC system in exploiting massive parallelism at the disk level. Configuration HC-1-6-0 shows the role that processors in memory play in queries that rely on combining operations. Configurations HC-1-2-4 and HC-1-1-6 were selected to show the behavior of more balanced HC systems.

6.2.1 Data Transferred

Our analysis of query execution revealed that the amount of data transferred over the processor to memory interconnect is the limiting factor for data access time, which in turn limits performance. Thus we look first at how HC systems can reduce this data transfer. Figure 6.1 shows the amount of data transferred over the processor to memory interconnect for each configuration. This includes the initial transfer from disk to memory, and then the transfer of data from memory to the caches. Depending on the nature of the query, intermediate tables may be created and held temporarily in memory. And depending on the size of the tables being accessed, groups of records may need to be moved and reloaded from disk.

We observed in Section 1.2 that on queries Q6 and Q14, conventional architectures transfer more than 4 times the amount of data required by the query. Even though Q6 performs a simple select operation accessing less than 25% of the columns of the largest table, the complete table still must be loaded to the memory. Additionally, Q14 loads another table and then builds three

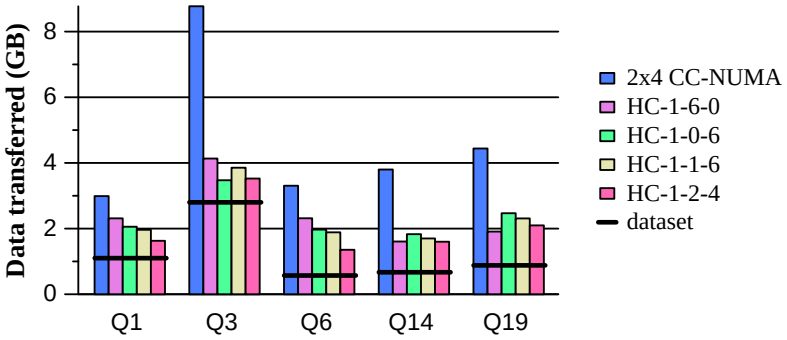

Figure 6.1: Amount of data transferred over the global interconnect for selected queries.

temporary tables.

In the HC configurations, this is performed within the disk. Hence significantly reducing the amount of data transferred outside of the node. As shown in Figure 6.1, the HC systems reduce the amount of data transfer by 37-58%.

6.2.2 Interconnect Contentions

As the amount of data transferred over the interconnect increases, the possibility of contention for the channel also increases. Contention occurs whenever a bus master agent (processor or DMA controller) is prevented from transferring data because the channel is being used to satisfy another request. Since we support a split memory transaction mode, two transactions can involve the same bus master.

Figure 6.2 shows the average number of processors waiting for the interconnect in a given cycle. This number grows considerably as the number of memory operations increases; it reaches its peak in query Q3, where the CC-NUMA system configuration averages 89.1 million memory transactions per second. At that point an average of 0.7 processors are waiting for one of

the 2 memory buses every cycle. By comparison, HC systems exhibit far less bus contention. This is due to the hierarchical bus topology of the system, and to the local access of storage locations by the processors in a node, which is controlled by the execution model. Hierarchical computing systems can significantly reduce the amount of bus contention, even in situations that exhibit a high transfer rate. However, there are cases where HC systems exhibit a considerable number of contentions, as for HC-1-0-6 and HC-1-1-6, when the disk processors transfer large amounts of processed data to memory. This is particularly evident in query Q1, where an average of 0.23 processors are waiting for the bus per cycle. However, even here bus contention is much reduced compared to the conventional configurations.

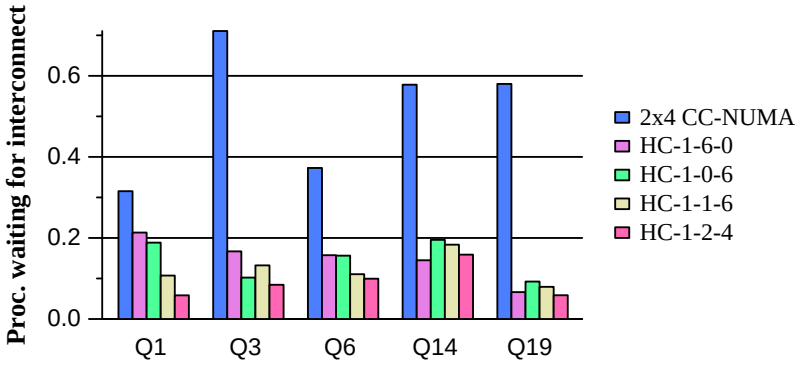

Figure 6.2: Average number of processors that are waiting for a global interconnect every cycle.

6.2.3 Average Memory Access Latency (AMAT)

A high number of contentions for the interconnect contribute to an increase in the average memory access latency. This is shown in Figure 6.3, where we define the average memory access latency as the average time a load instruction waits to receive its result.

One particular point of interest is seen in query Q14. For the 2x4 CC-NUMA configuration, a large portion of the L2 cache misses are serviced by the L2 cache of another processor within the node. But in our analysis of this configuration, we noticed that despite cache-to-cache transfers, the high volume of traffic made contentions the limiting factor of AMAT.

In the HC systems, we observe memory access times lower than the times in the corresponding CC-NUMA system. Note that the AMAT for the CC-NUMA configuration is higher than for the uniprocessor configuration, as a significant amount of the data used by the processors requires remote accesses (on average 32% of the total data transferred). These accesses impose an additional latency due to the coherence controller. The CC-NUMA system still achieves significant benefits, with speedups of up to $5.7x$ over the uniprocessor configuration (see appendix).

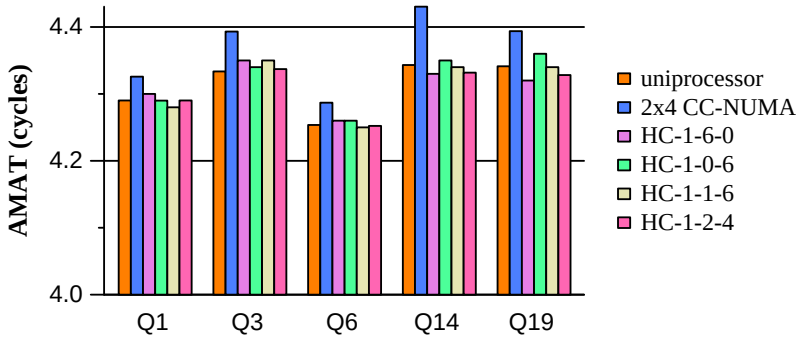

Figure 6.3: Average memory access time.

6.2.4 Speedup of Hierarchical Computing Systems

Figure 6.4 shows a comparison of the HC systems and a traditional multiprocessor system. System performance is shown with respect to the 2x4 CC-NUMA system configuration (2x4 CC-NUMA). The hierarchical computing systems outperform the equivalent CC-NUMA system on most of the queries.

The HC-1-1-6 configuration shows speedups between $1.10x$ and $1.20x$ when compared with the CC-NUMA configuration. The HC-1-2-4 configuration shows a slight slowdown of $0.98x$ in some of the queries, but also achieves speedups of up to $1.22x$. This slowdown is significantly small when we consider that it has 7 processors as opposed to the 8 processors of the CC-NUMA configuration.

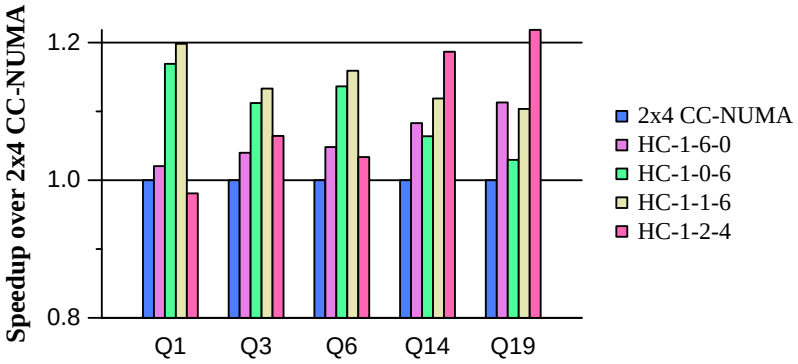

Figure 6.4: Speedups of hierarchical computing systems over base shared memory multiprocessor system with similar amount of computation and storage resources.

The selection queries (Q1 and Q6) are very regular and result in a relatively small number of interconnect contentions, as was shown in Figure 6.2. These transactions benefit from an increased number of processors, and are performed extremely well by the disk processors. Thus the HC-1-1-6 configuration outperforms the other systems. The HC-1-2-4 configuration has a reduced number of disk processors, which prevents it from achieving the high speedups of the other configurations.

Queries Q14 and Q19 consist of a moderate to large amount of combining operations, which are assisted by memory processors. The HC-1-2-4 configuration shows the best performance, achieving a speedup of $1.18x$. Con-

figuration HC-1-1-6 also does well in this set of queries, where the memory processor gives a larger advantage over configuration HC-1-0-6 than the one obtained for the previous queries. The memory processors also help configuration HC-1-6-0, which achieves a speedup of $1.11x$ over the CC-NUMA configuration.

For the large combining query (Q3), the memory processors do not help configurations HC-1-2-4 and HC-1-6-0 as much as in the previous two. Instead, configurations HC-1-1-6 and HC-1-0-6 show the best results. The reason for this apparent contradiction is that even when query Q3 performs a join, the disk access constitutes a larger portion of the execution time. Thus processors in disk provide the largest benefit.

6.3 Performance of an HC software-only system

The execution model used in the HC system allows it to reduce the amount of data transferred. We are interested in knowing if the use of the programming model alone is sufficient to improve the performance of the queries. Figure 6.5 presents the speedups obtained for a software implementation of an HC system (2x4 CC-NUMA-HCsw). This configuration is implemented on top of a 2x4 CC-NUMA system, as described in Section 5.2.2. We compare it to the CC-NUMA configuration using a traditional execution model (2x4 CC-NUMA) and a full HC system (HC-1-2-4). While observing the performance of each configuration, we noted that the amount of data transferred in the HC software-only configuration is 7.5% to 14.7% greater than in the 2x4 CC-NUMA configuration. Since the configuration does not have the local buses provided by a full HC system, the increase in data transferred results in a larger memory access time. The use of a software HC in a CC-NUMA

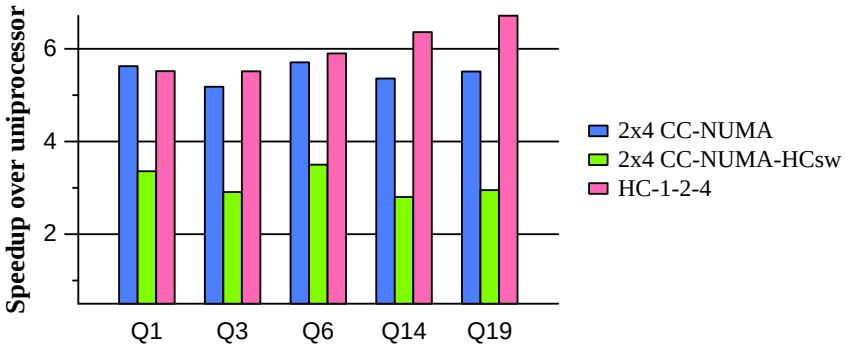

Figure 6.5: Speedups of a software implementation of a hierarchical computing system.

system also restricts the types of operations that a processor can execute. So it reduces the parallelism when compared with the conventional CC-NUMA system. In general, performance improvements come as a result of more efficient execution of the individual operations or due to an increase in parallelism. It does not come as a surprise then that for all the queries, the use of the software HC model on a CC-NUMA system results in a worse performance than a conventional CC-NUMA system. As can be seen from Figure 6.5, the smaller speedups with respect to the uniprocessor configuration come from those queries that have a heavy combining phase. This indicates that the factor most notably affecting the performance of this approach is the reduction of the parallelism when compared to a conventional execution model.

6.4 Processor Sensitivity Analysis

As indicated in Table 5.3, the processors in the memory and disk modules in the previous HC configurations are slower than the main processor. However, the comparisons are done with respect to a conventional system, whose processors are all *fast* processors. Thinking about this, an interesting

question is determining how much this decision affects the HC systems. After all, engineering tricks could allow HC systems to use faster processors as part of their memory modules. To answer this question, this experiment assumes that all the processors in the system have the same configuration as the main processor. That is, all the processors are out-of-order, can issue up to 4 instructions per cycle and operate at 1 GHz.

As shown in Figure 6.6, the benefit of having faster processors depends on the configuration being considered. There are configurations, such as HC-1-0-6, where there is no perceived difference. This result seems to indicate that the disk processors are fast enough to handle the data at the rate provided by the disks. Based on that observation, configuration HC-1-1-6 does not need faster processors in those queries that depend mostly on the disk processors. Figure 6.6 shows a very small difference for queries Q1, Q3, and Q6, which confirms the previous observation. Similarly Figure 6.6 shows that improving the processors in configuration HC-1-6-0 results in a large improvement for queries Q14 and Q19. These two queries depend on the processors in memory to perform a large portion of the join operation. This result also explains why improving the processors in configurations HC-1-2-4 and HC-1-1-6 results in some benefits for queries Q14 and Q19.

6.5 Scalability of HC Systems

Another important aspect of this study is to understand the benefits that HC systems can provide to multiprocessors of different sizes. When looking at large multiprocessor systems, we compare our results with a system with 32 processors (8x4 CC-NUMA). This system is organized as a CC-NUMA system with 8 nodes; each node being a 4-way SMP. The hierarchical computing

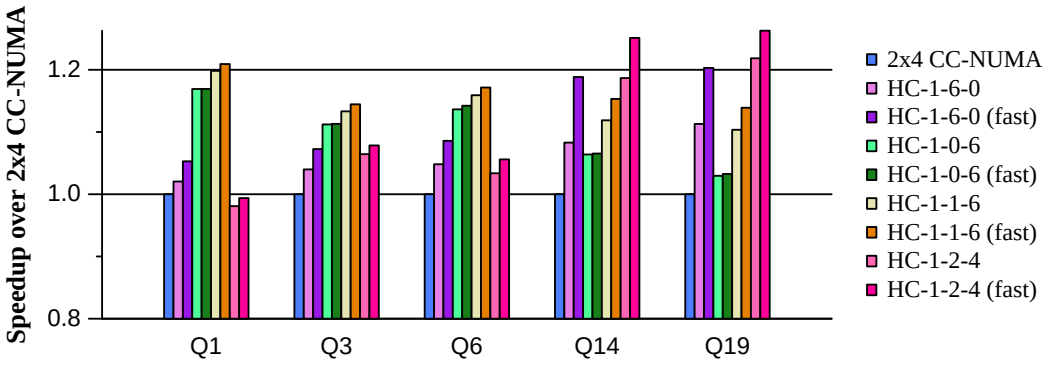

Figure 6.6: Speedups of hierarchical computing systems with fast processors.

systems (HC-1-5-25 and HC-1-6-24) have similar amounts of computation, storage and communication resources.

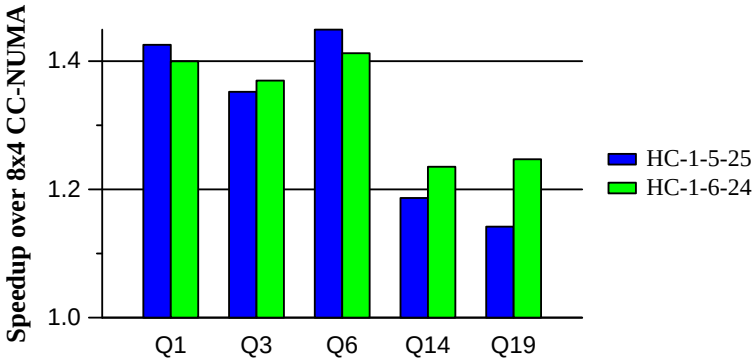

Figure 6.7: Speedups of large hierarchical computing systems over a base multiprocessor system with similar amount of computation, storage and communication resources.

We observe in Figure 6.7 that the HC systems outperform the CC-NUMA configuration in all the queries. The largest improvement was observed for the selection queries (Q1 and Q6), followed closely by the large combining query (Q3), which transfers the largest amount of data of all the queries. A significant but smaller improvement is seen in the combining queries. This diminishing return is due to the reduced number of computing elements used

to perform the combining component of the algorithm. Overall the HC systems exhibited speedups in the range $1.14x$ to $1.45x$.

We can also see that the HC systems show slightly different performance in the different queries. The HC-1-6-24, which has 6 memory-processor modules, is better for the queries with a high combining component (Q3, Q14, and Q19). The HC-1-5-25, which has one more disk-memory module, does slightly better in the queries with a high selection component (Q1 and Q6). However, the difference is not as marked as that between HC-1-0-6 and HC-1-2-4, as shown in Section 6.2.

To investigate smaller multiprocessor systems, a 4-way SMP system is used as the base configuration. Three HC configurations are selected: an HC

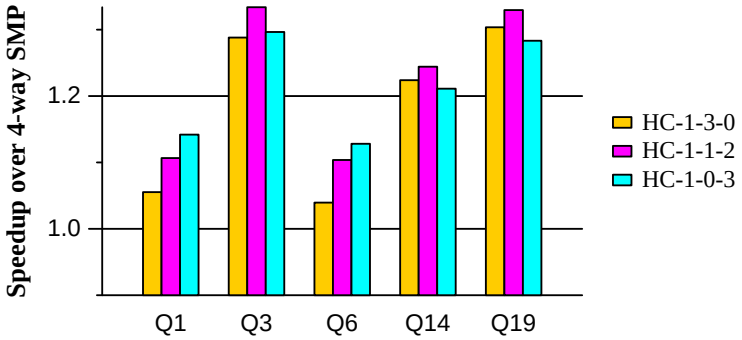

Figure 6.8: Speedups of small hierarchical computing systems over base shared memory multiprocessor system with similar amount of computation and storage resources.

system with 3 memory-processor modules (HC-1-3-0), an HC system with 3 disk-processor modules, and a balanced HC-1-1-2 system. We observe that the HC-1-3-0 configuration achieves speedups between $1.12x$ and $1.46x$ when compared with the SMP-4 configuration. As in the previous experiments, we observe that the memory-processor modules are especially useful for the combining queries (Q3, Q14 and Q19).

6.6 Task Mapping Heuristic Trade-offs

To test the performance of the task mapping heuristic, the algorithm operates on the group of queries described in Section 5.3. The tasks are formulated based on the query plans listed in Figures 5.4 to 5.8. The system nodes definitions correspond to those of the configurations used to evaluate the performance of the HC system (Section 6.1).

The mappings obtained using the task mapping heuristic are compared to optimal solutions. These solutions are computed by doing a deep search of the design space. A deep search evaluates every possible combination of the design variables and determines if it is a valid solution for the problem. This operation turned out to be extremely time consuming, requiring between 75-283 seconds to run in the system described in Table 4.3. Three factors are considered when evaluating the performance of the task mapping heuristic. The first is an accuracy metric defined as the percentage of times that the algorithm reaches the optimal solution. The second factor is an effectiveness metric, which indicates how close the cost of the solution is to that of the optimal solution. Even when the optimal solution is not reached, an effective heuristic would find a close substitute. We also look at the compute time, normalized to that of the deep search technique.

Table 6.1: Performance of the task mapping heuristic.

	Accuracy	Cost Error	Normalized Compute Time
Deep search	100%	0%	1
Heuristic	72%	0% - 18%	0.023 - 0.137

Table 6.1 shows the performance of the heuristics for the set of queries.

The deep search method arrives to the optimum solution all the time. However, it requires between 75 to 283 seconds to compute the solution. Given that all queries studied in this work run in under 175 seconds, this is an unacceptable cost. The heuristic reaches an optimal solution 72% of the time, and in those cases where it does not, the cost is up to 18% higher than the optimal. It requires from 2 to 14% of the time needed for a deep search.

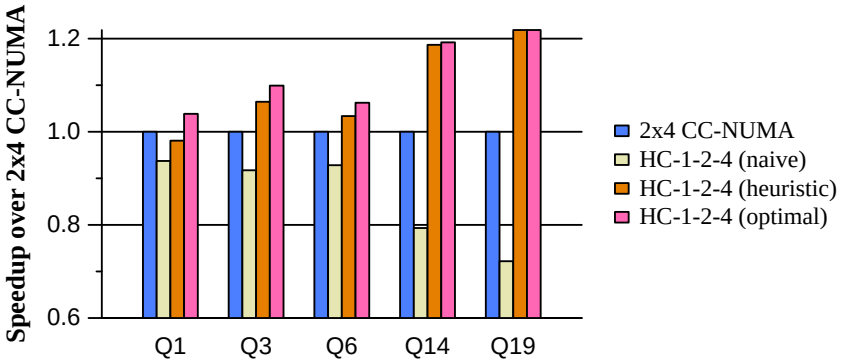

Figure 6.9: Speedup obtained by using the Task Mapping Heuristic.

The results presented in the previous section indicate that configurations with a large number of disk processors perform better in queries that have a large scan component. One mistake is to think this observation implies that the scan operation needs to be performed only by the disks to achieve a good performance. We refer to this policy as *naive task mapping*. The problem with this policy is that it does not considers the locality of the data for a given operation. So frequent scans to table are performed by the disk processor when they could have been done by a memory or main processor if the data were promoted to memory.

Figure 6.9 shows the speedups obtained for the HC-1-2-4 configuration over the CC-NUMA configuration. Three different task mappings are used. The first one (naive) assumes that all table scan operations are performed

by the disk processors. The second mapping is obtained by using the task mapping heuristic presented in this dissertation. These results have been presented in the previous section and are repeated here as a reference. The third mapping is the optimal mapping based on the cost minimization.

The results indicate that the naive mapping does not allow the HC system to compete against the conventional system. This gap is larger for the join queries, where a coordinated use of the processors is required to reduce the time of the query. For these queries, the HC configuration is 70 to 80% slower than the corresponding CC-NUMA configuration. This is a caused by poor use of data locality in the join operations when performing the scans completely at the disk level. The select queries (Q1 and Q6) are not affected to the extend of the join queries as the locality in these queries is reduced to local accesses of table segments, which is properly done by the disk processors. Figure 6.9 also shows how the heuristic approaches the optimal allocation in most queries.

Chapter 7

Data Placement Optimization

7.1 Introduction

Data placement is an important optimization done in conventional systems. The goal is to improve the performance, reliability or maintainability of the system by locating the data in a group of data storage devices (e.g., disk units). The performance of the system is usually improved by splitting the data across a large number of devices. This operation allows the system to perform multiple I/O access in parallel, thus increasing the I/O bandwith of the system.

This chapter formulates data placement as a combinatorial optimization problem and uses simulated annealing to arrive at a good solution for HC and conventional systems. Combinatorial optimizations are used in many branches of sciences to find the best arrangement of objects for a given constraint. For example, they are used in the context of VLSI and Electronic Design Automation to solve placement and route problems [78, 79]. Swanson et al. recently conjectured that SA could be used to assist instruction scheduling and place-

ment [86]. Similarly, we show that this technique is useful for determining server data placement.

7.2 Simulated Annealing

This chapter addresses the placement of data in the different nodes of a server. The data may be accessed by processors in local or remote nodes. Our main goal is to reduce the number of remote data accesses performed by those processors. The task of selecting the placement resulting in the lowest cost has been shown to be NP-complete for general graphs [26], so most algorithms use heuristics to approximate a solution.

Simulated Annealing (SA) [46, 1] is an algorithm for optimizing a solution over an enormous state space in a quick and effective manner. It does not guarantee the optimal solution, but given adequate parameters it produces a solution close to the global minimum in a fraction of the time it would take to do an exhaustive search. SA starts with a configuration and searches for better solutions by making random modifications to it (permutations) and evaluating their effect. If the permutation results in an improvement, it becomes the *current* configuration and the process continues. However, to escape local minima, the SA algorithm is allowed to accept some solutions that do not perform well.

The probability of accepting a harmful change depends on the cost it incurs, and the *temperature* T, an artificial parameter that controls the progress made by the algorithm.

$$prob(\Delta C, T) = exp(-\frac{\Delta C}{kT}) \tag{7.1}$$

The algorithm starts at a high initial temperature, T_0, and reduces the temperature at each iteration until it reaches the final temperature, T_f. The first

1) Set the initial solution; $S = S_0$.

2) Set the initial temperature; $T = T_0$.

3) Repeat while $T > T_f$:

   (a) For each permutation at this temperature step:

      i. Produce a new solution S_{new} by a random permutation of S.

      ii. Calculate the cost differential ΔC between S and S_{new}.

      iii. Set S to S_{new} if:

         - $\Delta C \leq 0$ or
         - $\xi < prob(\Delta C, T)$ for a random number ξ, $0 \leq \xi \leq 1$ and the temperature T.

   (b) Move to the next temperature step by reducing T according to the cooling schedule.

Figure 7.1: Simulated annealing algorithm.

iteration begins with a feasible (i.e., legal but not necessarily optimal), initial solution to the objective function, C_0. For each temperature step, 20 permutations to the solution are evaluated. If a permuted solution is better than the current solution, then it is immediately adopted as the current solution. If the solution is not better, it can still be adopted with $prob(\Delta C, T)$. This discourages solutions that settle early to a local minimum. Since $T_f = 0$, the algorithm halts when the probability of accepting worse solutions is zero. Figure 7.1 summarizes the general algorithm used for this study.

Figure 7.2 shows the cost obtained by only accepting solutions that reduce the energy (*Iterative Improvement* – II) versus the SA algorithm. The II technique is more likely to settle on a local minimum.

When applying SA to data placement, C becomes the effectiveness of

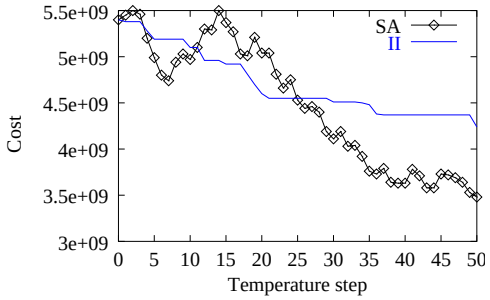

Figure 7.2: Cost function for the simulated annealing (SA) and the Iterative Improvement (II) algorithms.

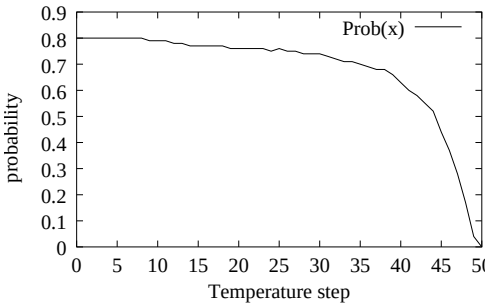

Figure 7.3: Probability of acceptance of a worse solution (uphill acceptance probability).

using a particular data placement. Example functions are the amount of inter-node data traffic or the time to complete the workload. The values for T progress from the high, initial temperature T_0 to the final temperature T_f, which is always zero. The values for k and T_0 are discussed in Section 7.3.3.

The reduction of temperature on each iteration is determined by the temperature cooling schedule. For all experiments, we reduce the temperature linearly. This results in the uphill acceptance probability shown in figure 7.3. It is possible to obtain better solutions by running the algorithm for more iterations. We use 50 temperature steps for the data placement.

7.3 Data Placement

To apply the simulated annealing technique to the data placement problem, we need to define a solution state, and objective function, and a temperature schedule. This section describes the process used to obtain those expressions using characteristics of the workload and the system.

7.3.1 Solution State

The solution state used in the SA algorithm describes the location of data in the server system. An individual data element can be represented by a tuple (D, L) that indicates the identity of the data and its location in the system. The solution state is the collection of the tuples for all the data.

However, this approach has two problems:

- variable-sized data elements could result in permutations that cause drastically different amounts of improvement in the solution.
- The larger the data element size, the more likely that many nodes will require data within the data element. This may actually raise the amount of inter-node data traffic.

To solve these problems, whenever possible, we divide the data elements into smaller, fixed-sized *chunks*. For our workloads, a chunk consists of multiple rows in a database table.

The initial solution state of the system is found by allocating the chunks sequentially across the storage elements in the system. The solution state is permuted by selecting a chunk and moving it to a different node in the server. The number of chunks determines the length of the data placement process. Later in the chapter, we explore the trade-off between the size of the chunks and the time it takes to run the algorithm and the solution's quality.

7.3.2 Objective Function

The goal of the data placement technique could be to improve the execution time, throughput, scalability, fault tolerance, or even power consumption of the server. The SA objective function must be selected to match this goal. In this work we use inter-node data traffic and execution time as objective functions.

Inter-node Data Traffic

Lovett et al. [54] showed that remote data accesses are one of the biggest performance bottlenecks in the execution of server workloads. Performance can be improved by a data layout that reduces the number of inter-node transfers. Therefore, we consider using inter-node data traffic as an objective function.

Data traffic in the system can be represented by a graph. The nodes in the graph represent the data storage and computation elements in the system, and arcs represent the transfer of data. Figure 7.4 shows the graphic representation of a sample problem. Computation is represented by P and data by D. Dotted lines represent node boundaries. The amounts of data transfer needed for each computation may differ. That information can be incorporated into the graph by assigning a weight to each of the arcs. The objective function is the sum of the weight of all arcs which cross the boundaries of a node. It is expressed by the formula:

$$cost = \sum a_{ij} w_{ij} \tag{7.2}$$

where

$$a_{ij} = \begin{cases} 0 & if\ node_i = node_j \\ 1 & otherwise \end{cases} \tag{7.3}$$

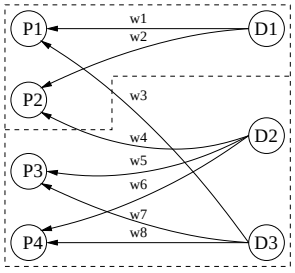

Figure 7.4: Data partition graph. The initial problem is to arrange the data between 2 nodes. In this configuration, the remote transfer cost is $C = w3 + w4$.

Time

We also consider an objective function expressed in units of time to run the workload. This function yields higher quality solutions, but the algorithm has a longer running time than the objective function based on remote data accesses. Cascaval et al. [14] have shown good results when estimating the time of scientific applications using a model of the form:

$$t_{total} \;=\; t_{cpu} \;+\; t_{mem} \;+\; t_{comm} \;+\; t_{i/o} \tag{7.4}$$

The CPU time is obtained using the number of instructions and estimated hardware costs. The memory time is obtained using a *stack distance model* [13]. The communication and I/O times are obtained from the amount of data that is transferred across the system's nodes or from the I/O devices. Equation 7.4 does not consider the overlap between these components, but it is sufficiently accurate for the purpose of this study.

7.3.3 Temperature Schedule for Data Placement

Since our application of simulated annealing does not correspond to any real physical phenomenon, we can substitute $k = 1$ in equation 7.1. We find the value for T_0 by using an initial uphill acceptance probability of 0.8 [45]

and solving equation 7.1 for T.

$$T_0 = -\frac{C_0}{ln(0.8)} \tag{7.5}$$

7.4 Evaluation Methodology

This section describes the hardware and software setup used for the experiments, and the procedure to obtain of the cost function.

The experiments simulate a 4x4 CC-NUMA configuration and an HC-1-3-12 configuration. The CC-NUMA server has four 4-way SMP nodes, as described in Section 5.1.1, for a total of 16 processors. The system has a total of 2 GB of memory and 28 disks. The HC-1-3-12 configuration has 1 main processor, 3 memory-processor modules and 12 disk-processor modules. Each disk-processor module has 2 disk units. There are also 4 disk units that hold the system image. Similar to the CC-NUMA configuration, this system has a total of 2 GB of memory and 28 disks.

We use the group of decision support system (DSS) queries presented in Section 5.3. The *time* cost function is obtained as in Section 4.3.2. The data transfer graph is used to generate the *data transfer* cost function as described in Section 7.3.2.

7.5 Results

Here we present the results of the simulated annealing method applied to the problem of placing data in a Hierarchical Computing system. It also shows the results obtained when applied to a conventional CC-NUMA system. We then examine the factors that affect the effectiveness of the technique.

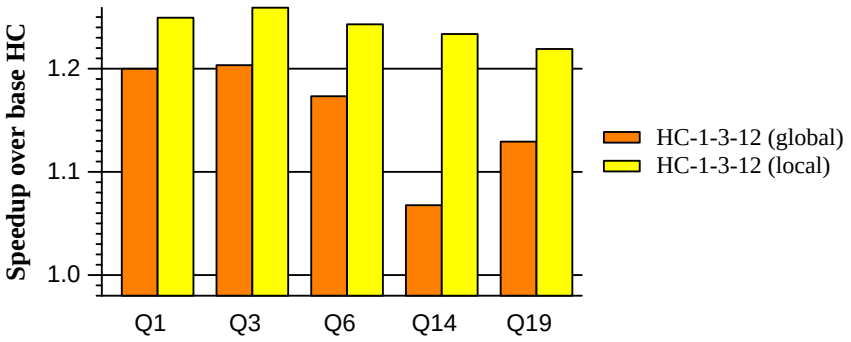

Figure 7.5: Performance of the simulated annealing data placement optimization on a Hierarchical Computing system.

7.5.1 Data Placement on a Hierarchical Computing System

To study how successful the SA technique is in providing an efficient data layout for the HC-1-3-12 system, we start with a layout used by the database to exploit intra-partition parallelism. This base layout spreads the data for each table equally among the data disks of the nodes using chunks of 1 MB.

The SA optimized layout uses 50 steps and data chunks of 16 MB and uses the technique described in Section 7.3. The *global training* method uses information on all the queries to produce a single layout for the system; the objective function is the sum of the inter-node data traffic for the 5 queries. Conversely, *local training* produces a layout optimized for a single query. For each layout, the experiment runs 2 queries of each type. The first is used to warm up the system; the required time is measured for the second query. Figure 7.5 shows the speedups of both SA layouts over the base layout.

The data placement optimization increases the performance of the HC-1-3-12 system by a factor between 7% and 20%, when using the global training. Local training increases the performance by a factor between 22% and 26%.

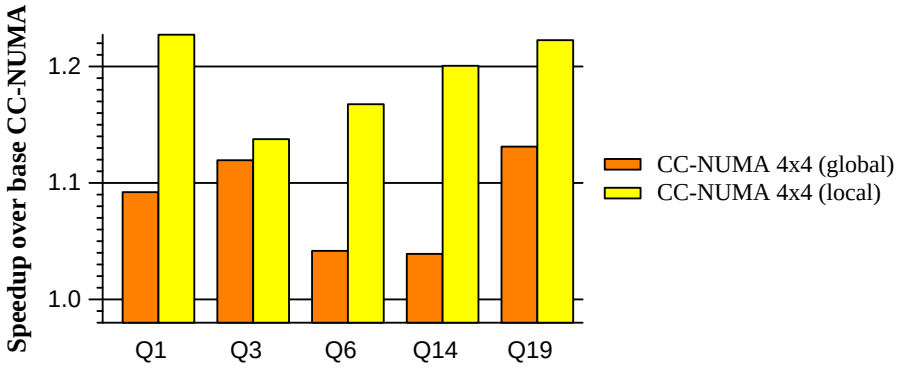

Figure 7.6: Performance of the simulated annealing data placement optimization on a CC-NUMA system.

7.5.2 Data Placement on a Conventional CC-NUMA System

We observe that SA produces layouts that effectively reduce the execution time of all the queries. The more practical global training produces speedups of 4 to 13%. The performance of all queries improve further with local training, where we obtain speedups of 14 to 22%.

Figure 7.7 shows a comparison of the HC-1-3-12 and 4x4 CC-NUMA configurations with and without the data placement optimization. While the base HC configuration displays speedups from 15 to 30% over the base CC-NUMA configuration, the optimized HC configuration shows speedups from 23 to 56%. This demonstrates the benefits of reducing data transfers for a DSS workload.

7.5.3 Sensitivity Analysis

In this section we analyze the factors that affect the performance of the SA data placement techniques.

Steps. The number of steps in the temperature cooling schedule greatly affects the quality of the resulting layout. The time it takes to generate each of these layouts is proportional to the number of steps used. The think time

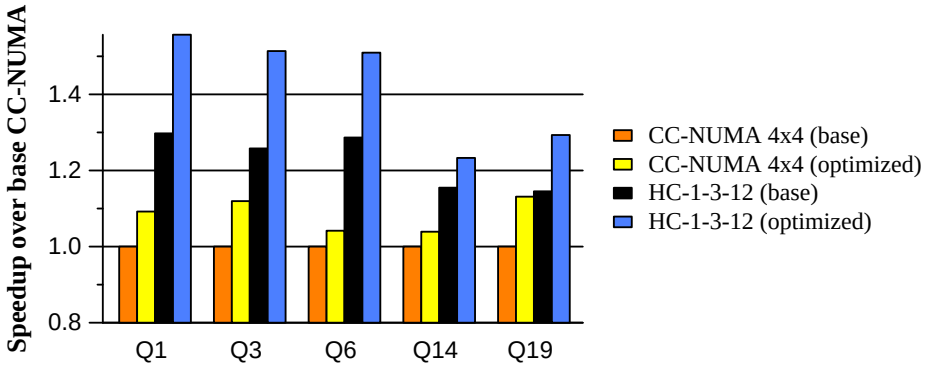

Figure 7.7: Performance comparison of the simulated annealing data placement optimization for HC and CC-NUMA systems.

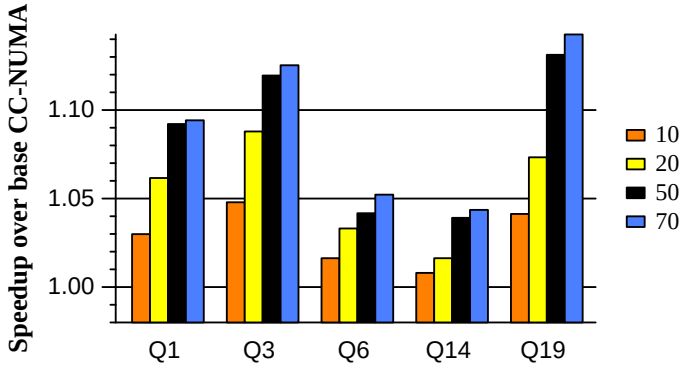

Figure 7.8: Impact of the number of steps over the simulated annealing process in the data placement optimization.

for a 50 step SA takes about 0.87 seconds of time on one processor. This experiment shows the performance for 10, 20, 50 and 70 steps.

Figure 7.8 shows the results of this experiment. We observe that increasing the number of steps improves the effectiveness of the layout. We do not see a significant improvement between 50 and 70 steps.

Chunk size. We also test the size of the chunks moved for each of the combinations. Using 50 steps and a process similar to the one described above, we change the size of the chunks from 4 MB to 32 MB. Changing the size of the chunks in our technique affects the number of elements that are

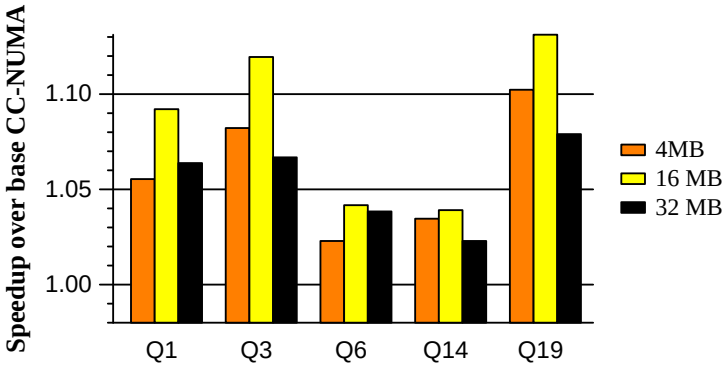

Figure 7.9: Impact of the chunk size over the simulated annealing process in the data placement optimization.

part of the optimization process. The more elements we have, the longer it takes to achieve a low energy configuration. Having large elements, on the other hand, reduces the flexibility of the layout. Figure 7.9 shows our results. We observe that all queries benefit from an intermediate chunk size. The appropriate chunk size needs to be determined for each problem. Empirically we found that an adequate estimate is given by:

$$k \times Steps \times Chunk\ Size \sim Data\ Size \tag{7.6}$$

for a value $1.5 < k < 2$. We also observe that join queries (Q3, Q14 and Q19) can benefit from smaller chunks.

Objective function. In these experiments we test which objective function is most effective regarding the SA technique. We use 50 steps and a chunk size of 16 MB. We pick functions that are related to the bottlenecks in the workload, so that reducing them will result in improved performance. These are described in Section 7.3.2. The first is the amount of traffic due to remote accesses (*inter-node*). The second reflects the time needed to execute the queries (*time*). Figure 7.10 shows the results of this experiment. We

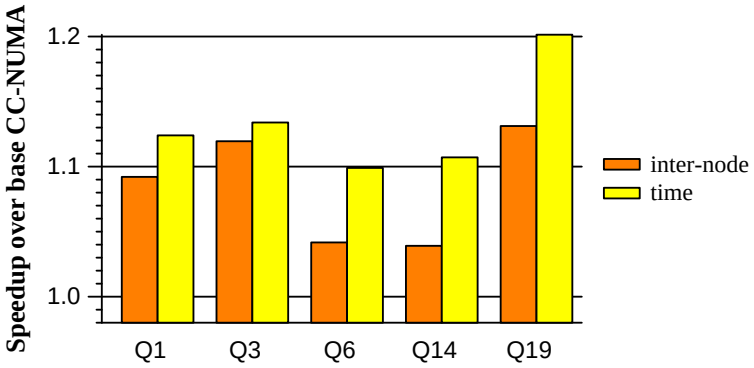

Figure 7.10: Performance of different objective functions in the simulated annealing process in the static data placement.

observe that it is beneficial to use a metric that more closely resembles the metric we shall eventually measure. The time objective function outperforms the inter-node objective function for all the queries. The only drawback is the time required to complete the optimization process. Optimizing the data layout using the inter-node objective function takes 0.87 seconds. When we use the time objective function, the system takes 2.13 seconds to generate the layout. Queries Q6, Q14 and Q19 show a significant improvement when using the time objective function, due to the computation intensive nature of the queries. Queries Q1 and Q3 are mostly memory bound, so the inter-node objective function is sufficient.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This dissertation looks at server systems running data intensive commercial applications. It addresses the problem of large data transfers in the system, which together with interconnect contentions limit the performance of the system. It proposes the Hierarchical Computing model as a solution to this problem.

The proposed system improves the performance of a commercial server system by providing: (1) a system that integrates processors across the memory/storage hierarchy; (2) an execution model to take advantage of this system; (3) a heuristic to map operations across the system; and (4) a heuristic to optimize the data layout to improve performance. This chapter presents the conclusions for this work and future directions for the design of Hierarchical Computing systems.

The Hierarchical Computing system places computing elements throughout the memory/storage hierarchy. These processors access data locally, thus

reducing data transfers over global interconnects. The system is designed on the assumption that a given operation may be performed more efficiently in disk, in memory, or in the central processor. An innovative software model allows us to distribute operations among computing elements which may be located anywhere in the storage hierarchy. One HC system can handle specific kinds of tasks better than a second, when each has same number of processors, depending on the placement of the processors. This work shows that systems integrating processors in these locations substantially outperform conventional systems.

This dissertation presents a heuristic that maps the operations of a task across the different nodes of an HC system. The heuristic is shown to result in task maps that have similar cost to an optimal map. Section 6.6 shows that computing the map using the heuristic requires between 2.3-13.7% of the time required to conduct a deep search.

We evaluate the HC model using full system simulation and a group of queries from the TPC-H benchmark. An HC system with 1 memory module and 6 disk modules performs better than an equivalent multiprocessors system in queries with a high select component (Q1 and Q6), where it exhibits speedups between $1.10x$ and $1.20x$. An HC system with 2 memory modules and 4 disk modules shows a small slowdown of $0.98x$ for the select queries (Q1 and Q6). This slowdown is surprisingly small, considering that the system has one less processor. In the combining queries (Q3, Q14, and Q19), the same system shows speedups of up to $1.22x$. The advantage of our computation model lies in the reduction of the amount of data transfers, which in turn reduces the amount of bus contention on the system.

Computer server systems used to run commercial workloads do not scale

well for a large number of processors. The HC computation model improves scalability to larger configurations. Comparing an HC system with 31 processors to an 8x4 CC-NUMA system shows speedups between $1.14x$ and $1.45x$.

A question that arises in the design of an HC system is where to place the processors. We observe that operations that benefit from a high level of parallelism, such as table scans (Q1 and Q6), are impacted most by processors in disk. On the other hand, operations that require a moderate to high amount of data combining, such as table joins (Q14 and Q19), benefit from the use of processors in memory. Operations that include phases of both types (Q3) benefit most from HC systems that have a balanced distribution of processors across the memory/storage hierarchy.

Results also indicate that configurations that have a large number of memory-processor modules can benefit from faster processors. This observation is clear once we consider that even when the processors are faster, the data access latency and bandwidth stay fixed. Thus in the case of disk-processor modules, processors that run at 250 MHz are sufficient to handle the data at the rate of the disk unit.

Another contribution of this dissertation is a data placement optimization for HC and conventional server systems. Commercial applications access large volumes of data. Because this data is spread over multiple storage locations, operations commonly perform a large amount of remote data accesses. A poor data layout can hurt the performance of the system. A technique that uses Simulated Annealing is proposed to efficiently place the data in a server. The feasibility and effectiveness of this approach is demonstrated for a conventional CC-NUMA and an HC server, both running DSS queries from the TPC-H benchmark. We test this data placement on a 4x4 CC-NUMA server

and find speedups of 4% to 13% when using a global training set. Similarly, this optimization improves the performance of an HC-1-3-12 system by a factor of 7% to 20% over an identical system that uses a standard data layout. As a result the HC-1-3-12 system reaches speedups of 23% to 56% when compared to the base fine-tuned 4x4 CC-NUMA with standard data layout. This demonstrates the combined benefits of reducing data transfers by placing data carefully and making good use of local computation.

In conclusion, we believe that hierarchical computing, which exploits parallelism, distributes computations, and reduces data transport requirements, is a desirable model of computation for future server systems.

8.2 Future Work

The approach of integrating computation across the memory and storage hierarchy opens new venues for future research in the areas of computer architecture and operating systems. The adoption of this technique would also spur research in the areas of databases and algorithms.

This dissertation evaluated the benefits of an HC system for a decision support workload, but a similar analysis could be applied to other workloads. Workloads that benefit from this system architecture would access large amounts of data. Their components could benefit from different processor micro-architectures (e.g., a fast superscalar processor versus a low-power in-order processor). They would be parallelizable at a thread level; however, workloads that are massively parallelizable do not require an HC system, as they could be effectively performed in a conventional parallel architecture. Relevant workloads include on-line transaction processing, biological research, and biometrics.

A key observation of this work is the reduction of data transferred globally in an HC system. Instead processors use local buses, which are usually shorter and operate at a lower voltage. Intuitively, this would result in a reduction of the average power consumed by the system. Furthermore, the use of slower processors in the integrated modules has the potential of reducing peak power consumption, while at the same time maintaining an adequate performance. A more through experimentation and analysis is required in order to understand these trade-offs.

The current evaluation uses commodity processors situated in proximity to memory modules and disk devices. Previous work has investigated the design and potential of integrated memory-processor and disk-processor devices. These devices can make use of the larger bandwidth present at the sense amplifier level, and lower access latencies. Future research should study the potential of these devices in the context of HC systems to evaluate the advantages of the close integration of the components.

This work presents one heuristic to map the operations across the processors of the hierarchy and a second heuristic to produce a data layout for the system. It is expected that a technique that combines the two processes in one heuristic could improve the performance of the HC system even further. Similarly, the current evaluation uses the query execution plan produced by the unmodified database manager system. Work could be done in query optimization and algorithm evaluation to produce a set of routines and QEPs customized for this new architecture.

The Hierarchical Computing model proves to scale upward very well. This work shows greater improvements for configurations with 32 processors than for configurations with 8 processors when compared with conventional

CC-NUMA systems of comparable size. In this evaluation, the factors affecting HC system performance (Chapter 6) were considered with respect to mid-size systems. It will be useful to evaluate them specifically with respect to large systems.

Appendix A

Parameters of Characteristic Time Functions

The following tables show the coefficients used in the characteristic time functions presented in Section 4.3.2. The first value corresponds to the row independent parameter ($t_{row\ independent}$), and the second value to the row dependent parameter ($t_{row\ dependent}$). The tables used to estimate the cost of a join operation have three parameters: a row independent, a parameter that depends on the number of rows in the outer table of the join, and a parameter that depends on the number of rows in the inner table.

Table A.1: Parameters for table scan #1.

	Processor		
Region	1 GHz	500 MHz	250 MHz
Fit in L1 cache	1.00e-2, 3.00e-7	1.60e-2, 5.40e-7	2.23e-2, 9.86e-7
Fit in L2 cache	1.30e-2, 8.40e-7	1.70e-2, 1.56e-6	3.13e-2, 2.90e-6
Fit in TLB	1.38e-2, 2.20e-6	1.73e-2, 2.98e-6	3.47e-2, 3.15e-6
Fit in memory	1.90e-2, 3.30e-6	1.99e-2, 4.67e-6	3.76e-2, 4.99e-6
Does not fit in memory	2.00e-2, 1.47e-5	2.18e-2, 1.59e-5	3.85e-2, 1.70e-5

Table A.2: Parameters for table scan #2.

	Processor		
Region	1 GHz	500 MHz	250 MHz
Fit in L1 cache	1.00e-2, 3.13e-7	1.60e-2, 5.59e-7	2.23e-2, 9.98e-7
Fit in L2 cache	1.30e-2, 8.59e-7	1.70e-2, 1.59e-6	3.13e-2, 2.93e-6
Fit in TLB	1.38e-2, 2.24e-6	1.73e-2, 2.99e-6	3.47e-2, 3.17e-6
Fit in memory	1.90e-2, 3.31e-6	1.99e-2, 4.67e-6	3.76e-2, 5.01e-6
Does not fit in memory	2.00e-2, 1.47e-5	2.18e-2, 1.59e-5	3.85e-2, 1.70e-5

Table A.3: Parameters for table scan #3.

	Processor		
Region	1 GHz	500 MHz	250 MHz
Fit in L1 cache	1.00e-2, 3.02e-7	1.60e-2, 5.43e-7	2.23e-2, 9.91e-7
Fit in L2 cache	1.30e-2, 8.41e-7	1.70e-2, 1.56e-6	3.13e-2, 2.91e-6
Fit in TLB	1.38e-2, 2.20e-6	1.73e-2, 2.98e-6	3.47e-2, 3.15e-6
Fit in memory	1.90e-2, 3.30e-6	1.99e-2, 4.67e-6	3.76e-2, 4.99e-6
Does not fit in memory	2.00e-2, 1.47e-5	2.18e-2, 1.59e-5	3.85e-2, 1.70e-5

Table A.4: Parameters for table scan #4.

Region	Processor		
	1 GHz	500 MHz	250 MHz
Fit in L1 cache	1.00e-2, 3.01e-7	1.60e-2, 5.42e-7	2.23e-2, 9.88e-7
Fit in L2 cache	1.30e-2, 8.41e-7	1.70e-2, 1.56e-6	3.13e-2, 2.90e-6
Fit in TLB	1.38e-2, 2.20e-6	1.73e-2, 2.98e-6	3.47e-2, 3.15e-6
Fit in memory	1.90e-2, 3.30e-6	1.99e-2, 4.67e-6	3.76e-2, 4.99e-6
Does not fit in memory	2.00e-2, 1.47e-5	2.18e-2, 1.59e-5	3.85e-2, 1.70e-5

Table A.5: Parameters for index scan #1.

Region	Processor		
	1 GHz	500 MHz	250 MHz
Fit in L1 cache	1.20e-2, 1.62e-7	1.82e-2, 2.69e-7	2.85e-2, 5.03e-7
Fit in L2 cache	1.45e-2, 4.01e-7	1.79e-2, 7.72e-7	3.21e-2, 1.46e-6
Fit in TLB	1.52e-2, 1.02e-6	1.88e-2, 1.62e-6	3.55e-2, 1.95e-6
Fit in memory	2.02e-2, 1.70e-6	2.13e-2, 2.72e-6	3.91e-2, 3.01e-6
Does not fit in memory	2.23e-2, 5.37e-6	2.33e-2, 8.20e-6	4.07e-2, 9.97e-6

Table A.6: Parameters for index scan #2.

Region	Processor		
	1 GHz	500 MHz	250 MHz
Fit in L1 cache	1.22e-2, 1.73e-7	1.85e-2, 2.81e-7	2.88e-2, 5.19e-7
Fit in L2 cache	1.46e-2, 4.15e-7	1.82e-2, 7.83e-7	3.24e-2, 1.50e-6
Fit in TLB	1.53e-2, 1.05e-6	1.89e-2, 1.67e-6	3.57e-2, 1.99e-6
Fit in memory	2.04e-2, 1.76e-6	2.15e-2, 2.77e-6	3.93e-2, 3.05e-6
Does not fit in memory	2.24e-2, 5.42e-6	2.37e-2, 8.25e-6	4.09e-2, 1.00e-5

Bibliography

[1] Aarts, E. H. L., and van Laarhoven, P. J. M. Statistical cooling: A general approach to combinatorial optimization problems. *Philips Journal of Research 40* (1985), 193–226.

[2] Acharya, A., Uysal, M., and Saltz, J. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, USA, Oct. 2–7 1998), pp. 81–91.

[3] Agarwal, R. C. A superscalar sort algorithm for RISC processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-96)* (Montreal, Quebec, Canada, June 4–16 1996), pp. 240–246.

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

[5] Albonesi, D. H., and Koren, I. STATS: A framework for microprocessor and system-level design space exploration. *Journal of Systems Architecture 45*, 12-13 (June 1999), 1097–1110.

[6] Arvind, and Nikhil, R. S. 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. Springer-Verlag, Berlin, Germany, 1987. Lecture Notes in Computer Science 259.

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

[8] Ben-Miled, Z., and Fortes, J. A. B. A heterogeneous hierarchical solution to cost-efficient high performance computing. In *8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)* (New Orleans, LA, USA, Oct.23–26 1996).

[9] Ben-Miled, Z., Fortes, J. A. B., Eigenmann, R., and Taylor, V. On the cost-efficiency of hierarchical heterogeneous machines for compiler- and hand-parallelized applications. *International Journal of Parallel and Distributed Systems and Networks* (1998).

[10] Bodorik, P., Riordo, J. S., and Jacob, C. Dynamic distributed query processing techniques. In *17th Annual ACM Conference on Computer Science: Computing trends in the 1990's* (Feb. 1989), pp. 348–357.

[11] Bruno, J., E. G. Coffman, J., and Sethi, R. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM 17*, 7 (July 1974), 382–387.

[12] Cao, Q., Trancoso, P., Larriba-Pey, J.-L., Torrellas, J., Knighten, R., and Won, Y. Detailed characterization of a quad Pentium Pro server running TPC-D. In *Proceedings of the International Conference on Computer Design (ICCD99)* (Austin, TX, USA, Oct. 10–13 1999).

[13] Cascaval, C. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing* (San Francisco, CA, USA, June 23–26 2003).

[14] Cascaval, C., Rose, L. D., Padua, D. A., and Reed, D. A. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing* (La Joya, CA, USA, Aug. 1999), pp. 365–379.

[15] Chapin, S. J. Distributed and multiprocessor scheduling. 233–235.

[16] Compaq Corporation. AlphaServer GS80, GS160 and GS320 systems technical summary. White paper, Feb. 2002.

[17] Daniels, D., Selinger, P. G., Haas, L. M., Lindsay, B. G., Mohan, C., Walker, A., and Wilms, P. F. An introduction to distributed query compilation in R*. In *2nd International Symposium on Distributed Data Bases* (North-Holland, Amsterdam, 1982), H. Schneider, Ed., pp. 291–309.

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

[19] Dell Computer Corporation. Dell PowerEdge 8450 specification. http://www.dell.com/downloads/us/pedge/pedge_8450.pdf.

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

[21] DeWitt, D. J. DIRECT - a multiprocessor organization for supporting relational data base management systems. In *Proceedings of the 6th Annual International Symposium on Computer Architecture* (Philadelphia, PA, USA, Apr. 23–25 1979), pp. 182–189.

[22] DeWitt, D. J. DIRECT - a multiprocessor organization for supporting relational database management systems. *IEEE Transactions on Computer 28*, 6 (June 1979), 395–406.

[23] Elliott, D. G., Snelgrove, W. M., and Stumm, M. Computational RAM: A memory-SIMD hybrid and its application to DSP. In *Proceedings of the Custom Integrated Circuits Conference* (Boston, MA, USA, May 1992), pp. 30.6.1–30.6.4.

[24] Emer, J., and Gloy, N. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual*

International Symposium on Computer Architecture (ISCA-97) (Denver, CO, USA, June 2–4 1997), pp. 304–314.

[25] Figueiredo, R. J. O., and Fortes, J. A. B. Impact of heterogeneity on DSM performance. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture (HPCA-6)* (Tolouse, France, Jan. 2000).

[26] Gavril, F. Some NP-complete problems on graphs. In *Proceedings of the 11th Conference on Information Sciences and Systems* (1977), pp. 91–95.

[27] Gharachorloo, K., Sharma, M., Steely, S., and Doren, S. V. Architecture and design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA, Nov. 13–15 2000), pp. 13–24.

[28] Hall, M., Kogge, P. M., Koller, J., Diniz, P., Chame, J., Draper, J., LaCoss, J., Granacki, J., Brockman, J., Srivastava, A., Athas, W., Freech, V., Shin, J., and Park, J. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the High Performance Networking and Computing Conference (SC'99)* (Portland, OR, USA, Nov. 13–19 1999).

[29] Hankins, R., Diep, T., Annavaram, M., Hirano, B., Eri, H., Nueckel, H., and Shen, J. P. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36)* (San Diego, CA, USA, Dec. 3–5 2003).

[30] Harris, J. A., and Smith, D. R. Simulation experiments of a tree organized multicomputer. In *Proceedings of the 6th Annual International Symposium on Computer Architecture* (Philadelphia, PA, USA, Apr. 23–25 1979), pp. 83–89.

[31] Hsiao, D. K., Ed. *Advanced Database Machine Architecture.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1983.

[32] Hsu, W. W., Smith, A. J., and Young, H. C. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal 40*, 3 (2001), 781–802.

[33] Hsu, W. W., Smith, A. J., and Young, H. C. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. *ACM Transactions on Database Systems 26*, 1 (2001).

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

[35] IBM Corporation. IBM p670 description. http://www.ibm.com/servers/eserver/pseries/hardware/midrange/p670_desc.html.

[36] IBM Corporation. SimOS PowerPC. http://www.research.ibm.com/arl/projects/SimOSppc.html.

[37] Intel Corporation. Intel IXP network processors. http://www.intel.com/design/network/products/npfamily/, Feb. 2003.

[38] Ioannidis, Y. E., and Wong, E. Query optimization by simulated annealing. In *Proceedings of the ACM SIGMOD International Confer-*

ence on Management of Data (SIGMOD-87) (San Francisco, CA, USA, May 27–29 1987), pp. 9–22.

[39] Jain, A. K., Murty, M. N., and Flynn, P. J. Data clustering: A review. *ACM Computing Surveys (CSUR) 31*, 3 (Sept. 1999), 264–323.

[40] Kang, Y., Huang, W., Yoo, S.-M., Keen, D., Ge, Z., Lam, V., Pattnaik, P., and Torrellas, J. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the International Conference on Computer Design (ICCD99)* (Austin, TX, USA, Oct. 10–13 1999).

[41] Keeton, K., Patterson, D., He, Y. Q., Raphael, R. C., and Baker, W. E. 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, June 27–July 1 1998), pp. 15–26.

[42] Keeton, K., and Patterson, D. A. Exploiting disk intelligence for decision support databases. In *Proceedings of the 3rd Workshop on Computer Architecture Evaluation using Commercial Workloads* (Jan. 9 2000).

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

[44] Keeton, K. K. *Computer Architecture Support for Database Applications.* PhD thesis, University of California, Berkeley, CA, USA, 1999.

[45] KIRKPATRICK, S. Optimization by simulated annealing - quantitative studies. *Journal of Statistical Physics 34* (1984), 975–986.

[46] KIRKPATRICK, S., GELATT JR., C. D., AND VECCHI, M. P. Optimization by simulated annealing. *Science 220*, 4598 (May 13 1983), 671–680.

[47] KOGGE, P. M. EXECUBE - a new architecture for scalable MPPs. In *Proceedings of the 23th International Conference on Parallel Processing* (North Carolina State University, NC, USA, Aug. 15–19 1994), pp. 77–84.

[48] KOGGE, P. M., SUNAGA, T., MIYATAKA, H., KITAMURA, K., AND RETTER, E. Combined DRAM and logic chip for massively parallel systems. In *Proceedings of the 16th Conference on Advanced Research in VLSI* (Mar. 27–29 1995), pp. 4–16.

[49] LEE, J., SOLIHIN, Y., AND TORRELLAS, J. Automatically mapping code on an intelligent memory architecture. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)* (Monterrey, Mexico, Jan. 19–24 2001), pp. 121–132.

[50] LEGARD, D. Server sales rise as users rebuild. http://www.infoworld.com/article/04/05/28/HNserversalesrise_1.html, May 28 2004.

[51] LIPOVSKI, G. J. Dynamic systolic associative memory chip. In *Application Specific Array Processors* (Princeton, NJ, USA, Sept. 5–7 1990), pp. 481–492.

[52] LIPOVSKI, G. J. Patent US4989180: Dynamic memory with logic-in-refresh. Jan. 29 1991.

[53] Lipovski, G. J. A four megabit dynamic systolic associative memory chip. *Journal of VLSI Signal Processing 4*, 1 (1992), 37–51.

[54] Lovett, T., and Clapp, R. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA-96)* (Philadelphia, PA, USA, May 22–24 1996), pp. 308–317.

[55] Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W. J., and Horowitz, M. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA 2000)* (Vancouver, BC, Canada, June 12–14 2000), pp. 161–171.

[56] Manegold, S., Boncz, P. A., and Kersten, M. L. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th Conference on Very Large Data Bases (VLDB2002)* (Hong Kong, China, Aug. 20–23 2002), pp. 191–202.

[57] Maynard, A. M. G., Donnelly, C. M., and Olszewski, B. R. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, USA, Oct. 4–7 1994), pp. 145–156.

[58] McErlean, F. J., Bell, D. A., and McClean, S. I. The use of simulated annealing for clustering data in databases. *Information Systems 15*, 2 (1990), 233–245.

[59] Memik, G., Kandemir, M. T., and Choudhary, A. Design and evaluation of a smart disk cluster for DSS commercial workloads. *Journal of Parallel and Distributed Computing 61*, 11 (Nov. 2001), 1633–1664.

[60] Menasce, D., and Almeida, V. Cost-performance analysis of heterogeneity in supercomputer architectures. In *Proceedings of Supercomputing '90* (Nov. 12–16 1990), pp. 169–177.

[61] Michael, M. M., Nanda, A. K., Lim, B.-H., and Scott, M. L. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)* (Denver, CO, USA, June 2–4 1997), pp. 133–143.

[62] Miller, L. J. A heterogeneous multiprocessor design and the distributed scheduling of its task group workload. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, TX, USA, 1982), pp. 283–290.

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

[64] Motorola Inc. C-3e network processor data sheet. http://e-www.motorola.com/brdata/PDFDB/docs/C3ENPA1-DS.pdf, Nov. 2002.

[65] Nguyen, H., and Bozman, J. Worldwide server market shows second consecutive quarter of revenue growth, driven by volume server sales,

according to IDC.
http://www.idc.com/getdoc.jsp?containerId=pr2003_11_25_140001, Nov. 26 2003.

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

[67] Oskin, M., Hensley, J., Keen, D., Chong, F., Farrens, M., and Chopra, A. Exploiting ILP in page-based intelligent memory. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)* (Haifa, Israel, Nov. 16–18 1999), pp. 208–218.

[68] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. A case for intelligent RAM: IRAM. *IEEE MICRO* (Apr. 1997), 34–44.

[69] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. Intelligent RAM (IRAM): Chips that remember and compute. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)* (San Francisco, CA, USA, Feb. 6–8 1997).

[70] Patterson, D., Asanovic, K., Brown, A., Fromm, R., Golbus, J., Gribstad, B., Keeton, K., Kozyrakis, C., Martin, D., Perissakis, S., Thomas, R., Treuhaft, N., and Yelick, K. Intelligent RAM (IRAM): the industrial setting, applications, and architectures. In

International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97) (Austin, TX, USA, Oct. 12–15 1997), pp. 2–9.

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

[72] Pfister, G. F. *In Search of Clusters*, 2 ed. Prentice Hall, Upper Saddle River, NJ, USA, 1998.

[73] Rao, J., Zhang, C., Lohman, G., and Megiddo, N. Automating physical database design in a parallel database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-2002)* (Madison, WI, USA, June 4–6 2002), pp. 558–569.

[74] Riedel, E., Gibson, G., and Faloustos, C. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th Conference on Very Large Data Bases (VLDB'98)* (New York City, NY, USA, Aug. 24–27 1998), pp. 62–73.

[75] Răadulescu, A., Nicolescu, C., van Gemund, A. J. C., and Jonker, P. P. CPR: Mixed task and data parallel scheduling for distributed systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium* (San Francisco, CA, USA, Apr. 23–27 2001), pp. 39–41.

[76] Sahni, S. K. Algorithms for scheduling independent tasks. *Journal of the ACM 23*, 1 (Jan. 1976), 116–127.

[77] Samadzadeh, F. A., and Hedrick, G. E. A heuristic multiprocessor scheduling algorithm for creating near-optimal schedules using task system graphs. In *Proceedings of the Symposium on Applied Computing (ACM/SIGAPP)* (Kansas City, MI, United States, 1992), pp. 711–718.

[78] Schuler, D. M., and Ulrich, E. G. Clustering and linear placement. In *Proceedings of the 9th ACM/IEEE Design Automation Conference* (1972), pp. 50–56.

[79] Shahookar, K., and Mazumder, P. VLSI cell placement techniques. *ACM Computing Surveys (CSUR) 23*, 2 (June 1991), 143–220.

[80] Standard Performance Evaluation Corporation. SPEC CPU92 benchmark.
http://www.specbench.org/cpu92.

[81] Standard Performance Evaluation Corporation. SPECsfs93 benchmark.
http://www.specbench.org/sfs93.

[82] Stanley, T. J., and Mudge, T. Systematic objective-driven computer architecture optimization. In *Proceedings of the 16th Conference on Advanced Research in VLSI* (Mar. 27–29 1995), pp. 286–300.

[83] Stenstrom, P., Hagersten, E., Lilja, D. J., Martonosi, M., and Venugopal, M. Theme feature: Trends in shared memory multiprocessing. *Computer 30*, 12 (Dec. 1997), 44–50.

[84] Stets, R., Barroso, L. A., Gharachorloo, K., and Verghese, B. A detailed comparison of TPC-C versus TPC-B. In *Proceedings of the*

3rd Workshop on Computer Architecture Evaluation using Commercial Workloads (Jan. 9 2000).

[85] Sunaga, T., Miyatake, H., Kitamura, K., Kogge, P. M., and Retter, E. A parallel processing chip with embedded dram macros. *IEEE Journal of Solid-State Circuits 31*, 10 (Oct. 1996), 1556–1559.

[86] Swanson, S., Michelson, K., and Oskin, M. Configuration by combustion: Online simulated annealing for dynamic hardware configuration. ASPLOS Wild and Crazy III, Oct. 2002.

[87] Torrellas, J., Yang, L., and Nguyen, A.-T. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture (HPCA-6)* (Tolouse, France, Jan. 2000), pp. 15–25.

[88] Transaction Processing Council. Brief description of the TPC-A benchmark.
http://www.tpc.org/tpca.

[89] Transaction Processing Council. The TPC-B benchmark specification.
http://www.tpc.org/tpcb.

[90] Transaction Processing Council. The TPC-C benchmark specification.
http://www.tpc.org/tpcc.

[91] Transaction Processing Council. The TPC-D benchmark specification.
http://www.tpc.org/tpcd.

[92] Transaction Processing Council. The TPC-H benchmark specification.
http://www.tpc.org/tpch.

[93] Tsangaris, M. M., and Naughton, J. F. A stochastic aproach for clustering in object stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-91)* (Denver, CO, USA, May 29–31 1991), pp. 12–21.

[94] Tsangaris, M. M., and Naughton, J. F. On the performance of object clustering techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-92)* (San Diego, CA, USA, June 1992), pp. 144–153.

[95] Uysal, M., Acharya, A., and Saltz, J. H. Evaluation of active disks for decision support databases. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture (HPCA-6)* (Tolouse, France, Jan. 2000), pp. 337–348.

[96] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (Queensland, Australia, May 19–21 1992).

[97] WINTER, R. The growth of enterprise data: Implications for the storage infrastructure. *in Whitepaper Winter Corporation* (1999).

[98] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., , AND GUPTA, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA-95)* (Santa Margherita Ligure, Italy, June 22–24 1995), pp. 24–36.

[99] YANG, T., AND GERASOULIS, A. A fast static scheduling algorithm for DAGs on an unbounded number of processors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, NM, USA, 1991), pp. 633–642.

Index

Vita

Juan Carlos Rubio was born in Lima, Peru on July 8, 1973, the son of Juan Manuel Rubio and Lilibeth Sucre de Rubio. After living in Lima for 6 years, he moved to Panama, Panama, where he attended Colegio San Agustin. He entered the Universidad Santa Maria La Antigua in Panama, Panama in 1991, and received a Bachellor of Science in Electrical Engineering in July 1997. In August 1997 he entered the Graduate School at The University of Texas at Austin. Here, he obtained a Master of Science in Electrical and Computer Engineering in May 1999, after conducting research on the architectural characterization of Java bytecodes. During the summers of 1999 and 2000, he interned at IBM Austin Research Lab working on simulation and modeling of server systems. His research interests include computer system architecture, microprocessor architecture and operating systems. He was a student member of IEEE, IEEE Computer Society, ACM, and ACM SIGARCH.

Permanent Address: P.O. Box 6-5005
Panama, Rep. of Panama

This dissertation was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.