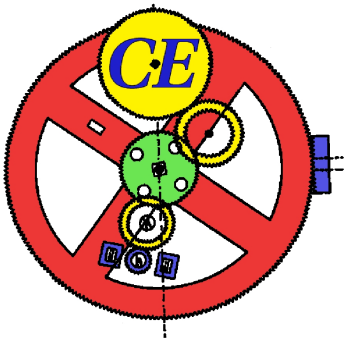


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Scalable Multi-core Architectures for Data Mining Applications

Madhavan Manivannan

CE-MS-2009-32

Abstract

Over the past few decades we have witnessed an exponential growth in the amount of data being used in virtually all domains. Extracting useful information from massive amounts of data is crucial and hence lot of data mining applications employing complex algorithms have been developed to aid in this task. As these data mining applications are gaining prominence it is very important to understand and address these application needs and translate them into design choices to have efficient and scalable processor architectures. This thesis is a step in this direction and involves investigating architecture alternatives that offer maximum scalability for data mining applications. For investigation we choose the Minebench benchmark suite from the data mining application domain and specifically worked with clustering and classification benchmarks. For evaluating the different architectural alternatives, we employ the SESC simulator. We identify the different architectural alternatives to be used for exploration by carrying out performance analysis experiments and identifying characteristics common to all the applications in this domain. Based on these observations, we select Asymmetric Chip Multi-Processor and Heterogeneous Multi-core Processor as candidate architectures and evaluate them by comparing them to a baseline Homogeneous Chip Multi-Processor. The evaluation shows that Asymmetric Chip Multi-Processor architectures provide better performance scalability than Homogeneous Chip Multi-Processor architectures for all the benchmarks considered. For Heterogeneous Multi-core Processor architectures we use two different scheduling strategies and observe large differences in performance results. Based on the results we conclude that Asymmetric Chip Multi-Processors consistently perform better than Heterogeneous Multi-core Processors.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Scalable Multi-core Architectures for Data Mining Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Madhavan Manivannan
born in Mumbai, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Scalable Multi-core Architectures for Data Mining Applications

by Madhavan Manivannan

Abstract

Over the past few decades we have witnessed an exponential growth in the amount of data being used in virtually all domains. Extracting useful information from massive amounts of data is crucial and hence lot of data mining applications employing complex algorithms have been developed to aid in this task. As these data mining applications are gaining prominence it is very important to understand and address these application needs and translate them into design choices to have efficient and scalable processor architectures. This thesis is a step in this direction and involves investigating architecture alternatives that offer maximum scalability for data mining applications. For investigation we choose the Minebench benchmark suite from the data mining application domain and specifically worked with clustering and classification benchmarks. For evaluating the different architectural alternatives, we employ the SESC simulator. We identify the different architectural alternatives to be used for exploration by carrying out performance analysis experiments and identifying characteristics common to all the applications in this domain. Based on these observations, we select Asymmetric Chip Multi-Processor and Heterogeneous Multi-core Processor as candidate architectures and evaluate them by comparing them to a baseline Homogeneous Chip Multi-Processor. The evaluation shows that Asymmetric Chip Multi-Processor architectures provide better performance scalability than Homogeneous Chip Multi-Processor architectures for all the benchmarks considered. For Heterogeneous Multi-core Processor architectures we use two different scheduling strategies and observe large differences in performance results. Based on the results we conclude that Asymmetric Chip Multi-Processors consistently perform better than Heterogeneous Multi-core Processors.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-32

Committee Members :

Advisor:	Ben Juurlink, CE, TU Delft
Chairperson:	Kees Goossens, CE, TU Delft
Member:	Zaid Al-Ars, CE, TU Delft
Member:	Henk Sips, PDS, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

First of all I would like to thank Prof. Ben Juurlink for his guidance and support throughout the length of this thesis work. His suggestions on improving the content and his critical remarks on my writing helped me enormously in compiling this thesis.

I would also like to thank Cor and Sebastian for their valuable suggestions and help. I would also like to thank Judit Gimenez, Mauricio Alvarez from Barcelona Supercomputing Center for helping me with the tools.

I would also like to thank Prof. Henk Sips and Prof. Zaid Al-Ars for accepting to serve on my thesis defense committee. I would also like to thank Prof. Georgi Gaydadjiev for his support and motivation.

Finally I would like to thank my parents, my brother and all my friends for their encouragement and support.

Madhavan Manivannan
Delft, The Netherlands
November 30, 2009

1 Introduction

1.1 Trends in Architecture Design

Processor performance has made massive advancements in the past few decades. We have advanced to an age where we use notebooks priced at a few hundred euros, which can deliver performance at many orders of magnitude higher than that provided by massive supercomputers of the past. This leap in performance was evident when the petaflop barrier was first surpassed by the Road Runner Supercomputer at LANL [4]. This machine employs hundreds of thousands of high performance processors each capable of delivering performance of the order of few hundreds GFLOPS. Experts believe that if this burgeoning industry advances at the current rate exaflop barrier would be achievable within a decade's time [2].

This growth attributed to the technological and architectural advances, is also starting to pose myriad design challenges. For instance, the current rate of doubling of transistor count every 24 months in accordance with Moore's Law is causing enormous pressures on processor architects to make use of this resource efficiently as they are running out of design options. Consider the case of the MIPS R12000 processor which has been introduced more than a decade ago. It bears many architectural enhancements and was able to deliver very high performance. It had features like speculation, wide issue, out of order execution etc to name a few. It was considered as a complex architecture with features that delivered massive improvement in performance over the previous generation processors. The overall design utilized around 7 million gates and was manufactured in 0.18 μm process technology [20]. In retrospection, looking at the current state of the art processors, it is clear that there have not been much architectural advancement from a design perspective. We still retain most of the same architectural features inspite of having 2 billion transistors to design modern day processors (this translates to a staggering 250 fold improvement in the number of resources merely in terms of transistor count). Although we have made few advances like SMT, trace cache, ISA extensions etc., the fundamental improvement in uniprocessor design has remained almost the same leading to design stagnation. Sparking enhancements are rarely seen (except the growth in cache size for each generation) and this trend is bound to continue.

To cope with the performance requirements and also to ensure efficient utilization of resources, the industry as well as academia have diverged from the uni-processor framework and have adopted the multi-core paradigm. Although there is still some focus on uniprocessor design (as this is the fundamental building block of the modern microprocessor), most of the community has started to focus its attention on issues surrounding design of multi-core processors. Multi-core architectures come in two

variants, homogeneous CMP and heterogeneous CMP. A Homogeneous multi-core is a design variant in which a single core is replicated to achieve a multi-core configuration. Most multi-core architectures which are currently on the market, barring a few models like the Cell, basically belong to this category. The primary motivation for the industry to stick to this model is to ease programmability, compiler design, OS complexity and resource management related issues [19]. Homogeneous multi-cores help in achieving performance scalability through parallel execution of threaded workloads but it is still inferior in terms of performance offered at a per thread basis when compared to its heterogeneous counterpart [31].

Heterogeneous multi-core is being worked upon by researchers in academia as it has better potential and this is largely because of the improved power-performance ratio [29], that it offers. Its impact on future processors has been outlined in [28], but a lot of general issues need to be sorted out before it is adopted. Many interesting proposals along the line of heterogeneous CMP's have also been made viz. an asymmetric multi-core processor [35], Conjoined Core Chip Multi-Processor [30], Core-Fused Architecture [24] etc.

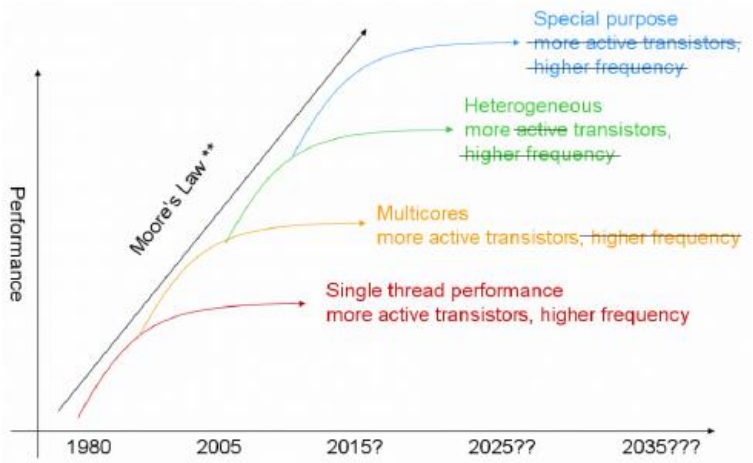


Figure 1.1: Trends in Architecture Design as a consequence of Moore's Law (taken from Hofstee [23])

Another promising paradigm involves specializing the design of multi-core processors to efficiently support the execution of certain class of applications, also known as 'multi-core specialization'. This specialization could range from simple architecture optimization that helps achieve scalable workload execution to application specific ISA extensions and even using hardware accelerators. The adoption of this paradigm could be mainly triggered by its enormous potential for power/performance improvement. To support this claim, let us consider the graph presented in Figure 1.1, taken from Hofstee [23]. The graph shows the general trend that designers have and will adopt in the coming years to address the issues related to power and performance. The graph

indicates that although the transistor budget keeps increasing, multi-core designers cannot have many operational transistors/cores if they want to stay under the power envelope. Under such a scenario core-customization would be the most convenient way to design architecture for future microprocessors [15].

In addition, advancements in application domain are also bound to have an impact on the design of future architectural platforms. Data Mining is one such application domain that is predicted to dominate the workload spectrum [12] and so it is crucial to explore systems architecture in view of efficient execution of these workloads. This offers a multitude of research challenges and opportunities and has got the attention of microprocessor vendors like Intel.

1.2 Architectures for Data Mining Applications

Data Mining essentially is the process of discovering Knowledge from large datasets and it has become an indispensable part in business and research alike. Its application ranges from identifying buyer patterns in supermarkets to genome sequencing. Many data mining techniques have been extensively applied in pattern recognition and in scientific applications in the HPC domain, which involve large scale data processing.

The growth of data mining is further fueled by Kryder's Law which in essence is very similar to Moore's Law and makes a prediction that the amount of bits that can be crammed on to storage devices would double annually [3]. In accordance with this law, storage capacity in scientific research institutes, business intelligence solutions and even desktop computers have seen an exponential growth having surpassed tera/peta bytes limits. This necessitates the design of algorithms to process large datasets efficiently to extract meaningful data and hence creating a need for faster execution of these workloads. This growth brings along ample opportunities for investigating various algorithmic and architectural optimizations for efficient execution of these workloads.

Current literature on accelerating mining workloads are classified into two broad categories: One that focuses on optimizations in the system architecture to considerably enhance performance, and the other that focuses on optimizing mining algorithms at large for faster execution on high performance microprocessors. For instance a few algorithms have been proposed recently, that have optimizations for achieving scalable performance on multi-core architectures. These include Cache-conscious prefix trees proposed by Ghoting et al. in [18] in which the tree structure proposed helps in improving benefits rendered by pre-fetching and a novel data structure that helps to achieve high temporal locality resulting in a speed up of 3.2 over current implementations. Buherer et al. in [11] proposed a parallel graph mining technique especially for CMP architectures that can adapt based on the runtime state of the system (a mechanism to vary memory consumption based on availability) and have found their approach to yield very good speedups of around 27 for a machine with 32 processors. This listing though not exhaustive, offers insights into some of the techniques adopted for workload

acceleration on emerging multi-core processor architectures by modifying the algorithm to take advantage of the architecture.

Prior works on optimizing systems architecture for data mining workloads mostly address the Cache/Memory Hierarchy and are based on the data access patterns of these applications. For instance, Ghoting et al. in [17] concluded that data mining applications exhibit poor temporal locality, Jaleel et al. in [25] identified that for some applications temporal locality across threads could potentially be exploited to improve cache hit rates by using shared last level caches as it helps increase the total storage capacity and Yi et al. in [13] used cache sensitivity analysis to show that cache hit rates can be improved by creating large lower level caches that can hold huge working sets and also proposed the use of technologies like 3D stacking to achieve large cache sizes. They conclude that exploiting spatial locality via hardware pre-fetching enables performance improvements for some applications. Kelly in [39], used architectural independent analysis of workloads to reconcile the discrepancy between Ghoting's, Yi's and Jaleel's work and showed that although temporal locality exists in these applications, most data experience very short periods of use separated by very long idle periods, i.e. while repeated use of data over the lifetime may encourage the use of large on chip caches to store complete working sets, the relatively short periods of activity requires carefully reconsideration of the proposal.

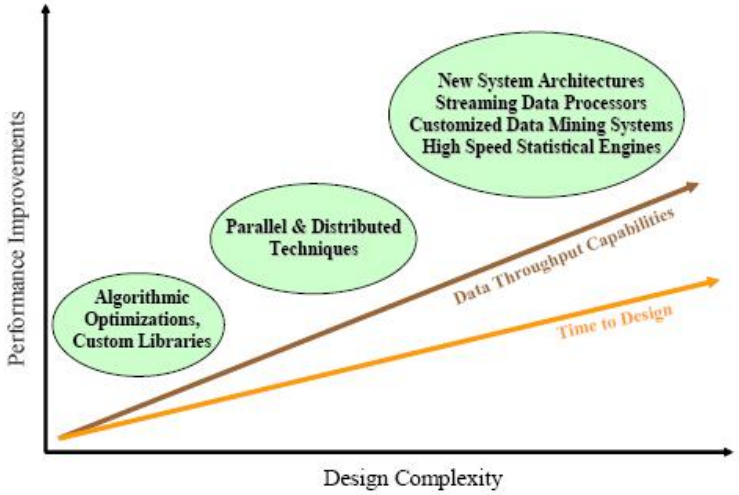


Figure 1.2: Tradeoffs across different optimization approaches (taken from Pisharath et al. [14]

In addition to Cache/Memory Hierarchy enhancements, Srinivasan et al. in [18] have proposed that SMT architectures are suited for some data mining applications because of the inherent data sharing nature that exists among the different threads. Recent work by Pisharath et al. in [14] presents the current approaches that have been adopted to accelerate data mining workloads. It also shows that there is lot

of research in direction of building custom designs and using reconfigurable logic for mining workloads. The graph presented in Figure 1.2 taken from the paper, clearly classifies the tradeoffs between the different optimization techniques. The authors clearly indicates that hardware based optimizations, although complex in terms of design, offer the best scope for performance improvements. This thesis explores a few architectural alternatives and investigates its potential for performance improvement.

1.3 Thesis Objective and Contribution

There is ongoing effort to improve the performance of data mining workloads on emerging processors, but a lot of possibilities and avenues remain to be completely explored. The objective of this work is to investigate architecture alternatives that offer maximum scalability during execution for majority of the applications in the data mining domain. This involves selecting candidate benchmarks, understanding architectural simulation platform, porting applications to use different threading libraries, investigating application scalability issues, identifying generic application characteristics, proposing alternatives to improve workload scalability on the basis of these characteristics and finally investigating the benefits of each of the architectural alternatives. Through this work, we have identified that the threading library used to parallelize data mining workloads plays significant impact in determining performance for communication intensive workloads. We have derived generic characteristics that are common across clustering and classification applications in the data mining domain. Also, we have evaluated architectural alternatives like Asymmetric Chip Multi-Processors and Heterogeneous Chip Multi-Processors for scalable execution of data mining applications and have found that Asymmetric Chip Multi-Processors performs consistently better than Heterogeneous Chip Multi-Processors.

1.4 Thesis Overview

Chapter 2 discusses about the issues pertaining to simulation and describes the various considerations that motivated the selection of a particular simulator. It also provides details on workload selection and its importance in the context of simulation and architecture evaluation. In Chapter 3, we discuss the impact of threading libraries on application scalability. Also, the scalability analysis results obtained by running these benchmarks on a SMP machine are presented. Based on these experiments we identify generic characteristics that are common across all the applications considered. Chapter 4 discusses the different architectural alternatives that are to be evaluated for improving application scalability and also present the simulation results of the baseline homogeneous configuration. In Chapter 5 we present the results of the architectural enhancements and also study the impact on the performance in comparison to the baseline homogeneous configuration. Chapter 6 concludes the thesis and outlines future work that can be carried out in this direction.

Architecture Simulation and Benchmarking

2

In this chapter we discusses the issues pertaining to simulation and benchmark selection for evaluating architectures. In Section 2.1, we provide an overview of architecture simulation. In Section 2.2, we discuss the classification of simulators based on evaluation capability. Then in Section 2.3, we discuss the challenges pertaining to computer architecture evaluation by using simulation. In Section 2.4, we highlight the simulation requirements for this work and in Section 2.5 we compare a few simulators to ascertain the best simulator that suits our requirement. Finally in Section 2.6, we discuss about the importance of benchmark selection and the choices we have made in this direction.

2.1 Computer Architecture Simulation - An Overview

The definition of simulation as termed in the dictionary is as follows: "The representation of the behavior or characteristics of one system through the use of another system, esp. a computer program designed for the purpose". The idea of building simulation model using computers is as old as the computer itself. Computer architects have always relied on simulation to make important design decisions. In today's microprocessor design cycle, the simulation phase plays a key role in determining the features that each successive processor generation must accommodate. With the current rate in proliferation of process technology, architecture with hundreds/thousands of cores will become a reality in a few years time. Having a fast simulation infrastructure to explore this vast design space in a reasonable amount of time is crucial.

Simulators are tools that assist the designers by allowing them to evaluate a multitude of design choices in a very short span of time under a single infrastructure, without having to make circuit level design and evaluation of every single feature. This improvement in turn around time for evaluating design choices has proved to be vital in current designs employing billions of gates which would otherwise have taken years to evaluate and also be severely cost prohibitive. For all the aforementioned reasons architecture researchers have increasingly relied on simulators to evaluate architectural enhancements. Lilja et al. in [44] have shown the trend followed for performance analysis by authors submitting their wok to ISCA to higlight the importance of simulation. Their results indicate that the use of simulation technique in papers (for evaluating architectural proposals) has grown from a modest 7.1% in the inaugural year this to around 90%. This clearly indicates the trust that architects have bestowed upon simulators and their role in the processor design cycle.

2.2 Classification of Simulators

The existing simulation ecosystem for architecture evaluation is explicated in this section. To help better understand the differences, a classification of the existing simulators is presented as shown in Figure 2.1. Before the discussion on classification, the difference between functional and timing fidelity needs to be understood. Functional fidelity refers to getting the execution correct whereas timing fidelity refers to modeling the execution time correctly. Mostly, simulators have this separated out in order to ease the task of development, as indicated in [5]. Simulators can be categorized as follows:

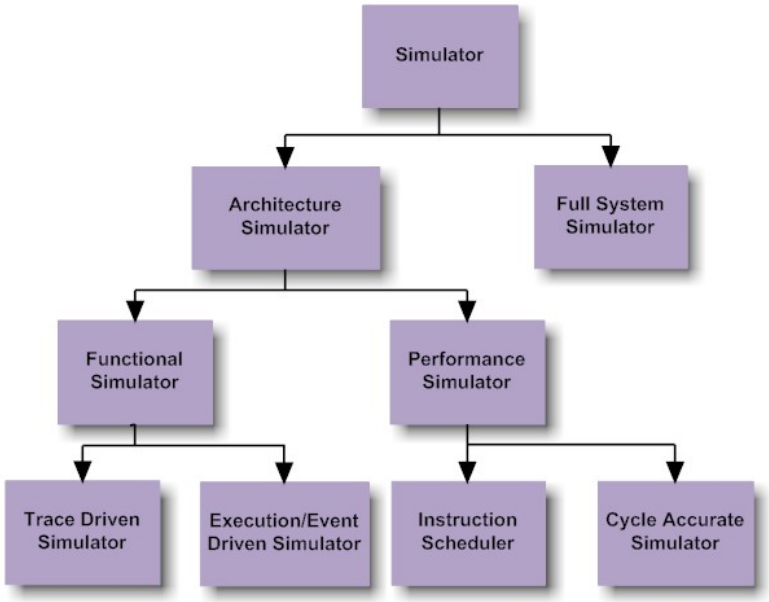


Figure 2.1: Classification of Existing Simulators

2.2.1 Full System and Architecture Simulators

A full-system simulator is generally able to model realistic workloads as it can support the design and debugging of hardware and software in the simulation framework. Operating System, database, device drivers etc. come under the category of software that the simulation infrastructure can support. This is generally used to evaluate software for new hardware platforms. To enable micro-architectural simulation, timing simulator models are generally integrated to the default functional simulation models.

An architecture simulator on the other hand, can model the architecture at great/abstract detail i.e. also at the level of instruction in a pipeline and also model speculative execution. These simulators can capture statistics of complex processor structures, but generally does not support Operating Systems. This is mainly because

the simulator has been designed with a focus on simulating individual processor components to collect statistics.

2.2.2 Functional and Performance Simulators

Functional simulators are faster as they only emulate the execution of an instruction. As a consequence they are less accurate and are hence generally not used to perform micro-architecture analysis. Typically, such simulation is performed to ensure software correctness and to also verify that applications, and computer programs that were designed for older, outdated hardware are properly executed on modern hardware. Performance simulators on the other hand allow to evaluate the time it takes for a system to execute the instructions of a given application. Performance simulators can further be classified into two distinct categories viz cycle accurate simulators and instruction schedulers.

A Cycle accurate simulator tracks the complete state of the micro-architecture every cycle. This methodology allows many instructions to co-exist in a single state/stage in a pipeline and modeling this aspect is required to establish accurate simulation of micro-architecture behavior. An Instruction Scheduler on the other hand is simpler to implement/modify as it is less detailed. As a consequence of its lack of detail it can schedule instruction based on resource availability generally in an inorder sequene.

2.2.3 Execution/Event Driven and Trace Driven Simulators

Simulators can be also classified as trace driven, execution driven or execution/event driven. Trace driven simulation works with instruction traces or memory references as input and is obtained by executing the simulated application. Traces are pre-recorded streams of instructions which allows for deterministic simulation each time and are commonly used for simulating memory hierarchies.

Execution driven simulator refers to the model in which functions are called to simulate the complete processor every cycle. Execution driven simulation is considered much more accurate than its trace driven counterpart because the former lets dynamic effects like synchronization and contention influence the simulated application's execution path and also models side-effects of the operating system. Execution/Event driven simulation is also a possible option in which functions are called to simulate some part of execution whereas the rest are modeled using event driven callbacks, i.e., a function call with parameters is scheduled, to occur at some time in the future.

Execution driven simulator can be either based on direct-execution or an interpreter model. In direct execution the application is directly executed on the host machine where the simulation runs. Since there is no simulated processor it can only be used in studies where there is no need for the processor to be modeled accurately. Few techniques to incorporate a processor model in order to improve timing accuracy have been proposed. Consequently, an Interpreter model is used when it is required to have a detailed model of

the architecture. In this model the simulator has to resort to interpreting the application executable thereby causing a considerable slowdown.

2.3 Simulation Challenges

2.3.1 Simulation Speed

Simulators have always been plagued by issues concerning the speed of simulation. Since simulators are seen as a representative model for real systems they are expected to run the same workloads as the actual system under consideration. But the speeds of current simulators are many orders of magnitude slower than the actual silicon. This mismatch in speed actually causes full sized benchmark programs to run for months together, a task which hardly takes seconds on a real machine. As a consequence architects have now realized that simulating benchmark programs for complete execution is a task that is virtually impossible. To remedy this issue, they have devised several innovative techniques which include typically simulating a sub-set of benchmarks, using reduced working sets or simulating a representative set of intervals from the program also known as sampling [44]. Among the aforementioned techniques sampling is found to be the most suitable in terms of simulation speed and accuracy as it effectively captures application behavior for a given input set using a smaller fraction of instructions.

2.3.2 Simulation Accuracy

The other aspect that influences every decision of the architect is the accuracy of the simulator. It is determined by how well the different parameters and their inter-relationships are modeled and how change in one parameter influences a change in another. Overall the accuracy is determined by the simulation methodology adopted, the representativeness of the benchmarks and the simulation technique adopted. As the design complexity grows, accuracy becomes all the more important because a small lapse in the part of the simulator might result in missing out on essential architectural enhancements.

2.4 Simulator Requirements

In addition to the general issues discussed in Section 2.3 that each simulator needs to address, there are additional requirements specific to the project that greatly influences the selection of a particular simulator. As there is a gamut of simulators available, the requirements have to be clearly laid down in order to help decide on the right platform for simulation. The following are some of the requirements that have been identified considering the scope of this project.

a) Multi-threading support: This is a very important requirement considering the scope of the evaluation to be carried out. Although there are lots of freely available simulators, most of them do not provide support for handling multi-threaded workloads. It must also be ensured that these multi-threaded simulators support the existing

threading libraries so as to enable rapid porting of application on the simulator.

b) Micro-architectural Simulation/Emulation Support: As the project involves investigating architectural optimizations it is crucial to have a simulator that can model micro-architectural parameters in great detail. In addition, the capability to model speculative execution at a uni-processor level is quintessential for accurate simulation, as indicated in [37]. Also, emulation support (instruction fast forwarding) greatly helps in reducing simulation time as it makes it possible to skip initialization and other un interesting sections in the application.

c)Portability/Support Issues: Many existing simulators were compiled for tool chains that are outdated and cannot be directly used with currently available tools. In some cases issues with respect to porting applications using the existing/modern tool chain can turn out to be difficult . Therefore tool chain support is quite important for a simulator. Also, attributes like developer support for a simulator, community mailing lists etc. can become quite important incase of working on simulators with potential bugs.

2.5 Overview of Existing Simulators

Although several simulator models exist in literature, this section only contains details on simulators that are widely accepted in the community and also discusses their applicability for this project.

2.5.1 SimpleScalar

The SimpleScalar tool-set is used for performance analysis studies, detailed micro-architectural modeling and hardware-software co-verification [5]. It includes several sample models suitable for performing a variety of architectural analysis. The simulators in the tool-set range from sim-safe, a minimal simulator that only emulates the instruction set, to simoutorder, a detailed micro-architectural model with dynamic scheduling, aggressive speculative execution and a multi-level memory system.

SimpleScalar is an execution driven simulator which requires the inclusion of an instruction-set emulator and an I/O emulation module. The instruction set emulator interprets each instruction directing the hardware models activities through interfaces that it provides. The I/O emulation module provides simulated programs with access to external input and output facilities. At the center of each model is the simulator core code which defines the hardware model organization and instrumentation. The simulator core defines the simulators main loop which executes one iteration for each instruction until they eventually complete. The performance core is comprised of baseline modules that make up the software architecture. These modules export functions ranging from statistical analysis, event handlers and command line processing to implementations of modeling components such as branch predictors, caches and instruction queues. Although SimpleScalar is useful for evaluating uniprocesors, it does not support simulation of multi-threaded workloads.

2.5.2 RSIM

RSIM is a publicly available tool for simulating shared-memory systems built from processors that utilize ILP features [37]. The authors have identified that modeling ILP features in a multiprocessor environment is important for applications that exhibit parallelism among read misses as it greatly helps in improving simulation accuracy. RSIM consists of several interchangeable modules to model a range of architecture. RSIM models processors that can exploit ILP by modeling processors with features like single-instruction issue, in-order instruction scheduling, multiple issue, out of order execution and non blocking memory operations. Also it provides a large list of user configurable parameters like issue width, window size, number of functional units etc.

RSIM provides a two-level cache hierarchy with separate L1 data and instruction cache and a unified L2 cache. It can be used to simulate several hardware variations, cache coherence NUMA architecture. Further details on modeling a multi-processor system with cache hierarchies and coherence mechanisms can be referred from [37]. The main drawback with RSIM is that it does not support CMP simulation. Also, the simulator is highly complex in terms of the effort required to understand and modify. The other factors which also don't favor RSIM include simulation speed and accuracy, i.e. RSIM is very slow for modeling out of order processors and the results have not been formally validated as well.

2.5.3 SIMICS

SIMICS [33] provides a full system simulation platform that is designed to run unmodified Operating Systems, real workloads, database benchmarks and other interactive applications. It can be used to model embedded system, desktop and even a multiprocessor system. In addition to this flexibility, it can also support a broad range of tasks throughout the product development cycle including microprocessor design, operating systems development, fault injection studies and hardware design verification. It currently supports models for UltraSparc, Alpha, x86, x86-64, PowerPC, MIPS and ARM. SIMICS includes an cache and IO timing model, allowing for a first order approximation of the memory operations and this can then be used as a platform for generating traces for the cycle accurate simulators. Also SIMICS processor models have been incorporated with features to support out-of-order processing but in a rudimentary way. Detailed micro-architectural studies can be carried out based on an arrangement that SIMICS would provide the functional model required for investigation and the user has to extend it with timing model. A number of timing simulators have been proposed and they can be used extend SIMICS to perform accurate micro-architecture simulation. Few interesting proposals based on extending SIMICS simulator for performing micro-architecture analysis are presented below.

a)GEMS: GEMS [34] leverages the existing simulation infrastructure and builds a set of timing simulator modules for modeling the timing of memory system and microprocessors. This approach led to reduced development time as a pre-existing full system simulator was used as a foundation on which only the timing modules had to be

dynamically loaded. This approach follows the timing first simulation model in which the functionality and timing aspects remain decoupled and the timing model interacts with the functional model to indicate when an instruction has to complete execution even though the execution is carried out by the functional simulator. At the core of GEMS is the RUBY memory simulator. If timing evaluation needs to be carried out using a simple inorder processor model provided by SIMICS, all loads and stores would would be forwarded to RUBY which would simulate the first level cache and then return if it is a hit, thereby enabling SIMICS to continue with the execution. In case of a miss, RUBY stalls SIMICS and then simulates a cache miss. For capturing the timing effects in dynamic superscalar processors it makes use of a processor model called OPAL which is based on timing first simulation model. This model works as follows: OPAL models the processor and determines when the instruction has to retire, it makes this decision when an instruction has reached the retire stage and instructs SIMICS to advance by one clock cycle. The state of the OPAL and SIMICS are then compared to check if the instruction has executed correctly.

b)SIMFLEX: SIMFLEX [21] presents a framework to arrive at fast and accurate simulation results for multiprocessor platforms by integrating the SMARTS methodology for representational simulation sampling. It uses a novel implementation technique for eliminating the runtime overheads that arise from component based software construction. SMARTS is a technique that can be used to accelerate simulation by only taking into account a subset of the benchmark. This subset is derived by a prescribed statistical procedure for configuring a systematic sampling simulation run to achieve a required confidence in the estimates. SMARTS assumes support for detailed simulation as well as functional simulation. In detailed mode all the relevant micro-architectural states are updated every cycle and in the functional mode only the correct state is maintained. SMARTS uses these two modes to sample at regular intervals, performing detailed simulation for sampled instructions and functional simulation for the rest. SIMFLEX succeeds in successfully applying this technique in a multi-processor context which comprises multiple instruction streams with asynchrony and non-determinism among them.

Both GEMS and SIMFLEX provide lot of leverage for multi-threaded simulation as demanded in the context of the project but were not chosen due to the characteristics of the underlying simulation framework. Although they can perform micro-architectural simulation at an acceptable level of accuracy, both these approaches also consider the OS overhead as part of the workload and incorporate this in the timing behaviour. Also extracting the timing profile for workload excluding the intervention of the operating system is also not feasible. This does not sync well with our initial goal of optimizing architecture purely considering the workload behaviour. The non-deterministic nature of the OS tasks could make it more challenging forcing to miss out on enhancements that can have potential impact on the performance.

2.5.4 SESC

SESC [38] is a micro-architectural simulator that is capable of modeling variety of architectures like uni-processors, Chip Multi-Processors and Processor In Memory. It can model an out of order pipeline with branch prediction, caches, buses and other components necessary to completly model a modern processor accurately. SESC is an event driven simulator built from the [42] emulator. The functional core of the simulator is execution driven whereas many required functions are called as and when needed, using events. Since SESC does not provide an Operating system, it traps system calls and performs them on behalf of the application. Every standard system call is transformed into a MINT function and is simulated by MINT. libapp is the application interface that SESC provides to emulate Pthreads and is also responsible for implementing locking API. SESC is preferred over all the previously discusses simulators because it can model CMP architecture by running muli-threaded workloads and supports micro-architectural simulation/emulation support without incurring the overhead of OS. Moreover SESC is very fast and is easily extensible to be able to test architectural enhancements. Although SESC has never been formally validated [16], it has been widely used as a tool to simulate multi-processor and CMP architectures. In addition, there have been numerous publications in top-tier conferences using this simulator.

2.6 Benchmark Selection

The first step in choosing the right set of benchmarks lies in identifying a proper benchmark suite. From the myriad benchmark suites available, one can either choose a benchmark suite that is not specific to any particular application domain like SPEC, PARSEC etc. or choose a suite which is oriented towards a particular domain like Alpbench, Mediabench, Minebench etc. As this work involves investigating architectural optimizations for data mining workloads we choose the Minebench suite [36]. The reasons that mainly motivate the selection of Minebench are as follows.

a)Multi-threaded Support: Multi-core architectures have become ubiquitous. Evaluating these architectures with a view of optimizing them for power/performance requires multi-threaded workloads that can investigate scalability issues and identify potential bottlenecks in architecture. Minebench provides a set of data mining benchmarks, comprising of application from Clustering, Classification and Association Rule Mining fields. These benchmarks have unique algorithmic features and is widely used in industry and research. The benchmarks in this suite also provides multi-threaded support (parallelized using OpenMP).

b)Workload Diversity: The benchmarks should exhibit diverse characteristics with respect to scalability and the computations involved. We are not into investigating applications that are either massively parallel or serial in nature (as they would provide very little scope for improvement), but are interested in application whose scalability lies in between these two regions, where optimization can greatly benefit application scalability, as depicted in Figure 2.2. As for computations involved in the benchmarks

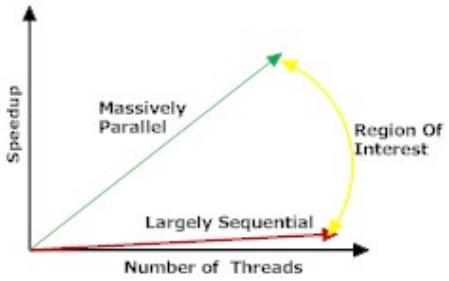


Figure 2.2: Region Of Interest.

(Minebench suite), Berkin et al. in [36] have show that the combination of high compute and memory intensity results in a clustering algorithm distinguishing data mining applications from existing benchmark suites. The results they have obtained (shown in Figure 2.3) indicate that these benchmarks are diverse (as they belong to different clusters). These results were obtained by performing statistical analysis on 19 architectural characteristics like branch prediction, L1 and L2 cache access etc. for each of the benchmarks and then clustering benchmarks with similar characteristics. The applications that we have chosen as candidates for evaluation (HOP, SCALPARC, KMEANS, FUZZYMEANS) fall under three different clusters.

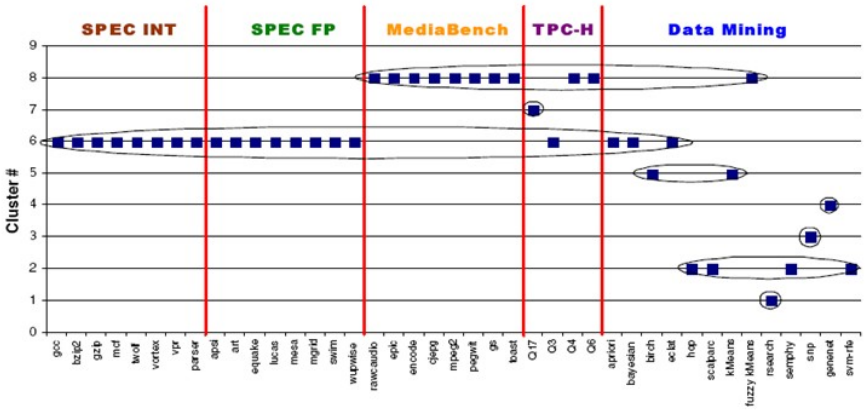


Figure 2.3: Classification of Benchmark Suites based on Architectural Characteristics (taken from Berkin et al. [36].)

There were many issues which prevented us from considering all the benchmarks in Minebench, for evaluation. For instance few of the benchmarks in the suite were serial (BIRCH, BAYSEIAN). A few others had limitations on the extent to which it could be parallelized (Apriori and Utility could not support more than 8 threads). A few others demanded the use of vendor specific libraries which were not available under open source license.

2.7 Conclusion

In this chapter, we first discussed the importance of choosing an appropriate simulation infrastructure. We analyzed a few existing simulators and we determined that SESC would be used for performing evaluation as it satisfied all the simulation requirements. Next, we discussed the impact of benchmarks on the overall performance results and also presented justification for using benchmarks from the Minebench suite.

Software Considerations and Workload Characterization

3

In this chapter we present the performance analysis results of the data mining workloads, discuss the impact of threading libraries on performance scalability and also identify generic characteristics that are common to all the applications. In Section 3.1 we first discuss the importance of threading libraries. Then in Section 3.2, we discuss the the issues involved in workload evaluation and we specifically consider evaluation platform, evaluation metrics and the evaluation tools . Following this, in Section 3.3 we discuss the performance analysis results of the workloads obtained by running the application on the candidate platform. Finally in Section 3.4 we list the observations that we have made regarding generic characteristics of the applications in data mining domain based on the performance analysis results and also conclude the chapter.

3.1 Programming Model and Threading Libraries

Although the architecture largely determines the performance of an application, the programming model also plays an equally important role in utilizing this performance by providing abstractions to represent the parallelism inherent in an application and in a manner that also matches the architectural model. While numerous models have been proposed, a few have gone on to become widely accepted (albeit with their own set of drawbacks). Among them are distributed memory/message passing programming model where each task is encapsulated with its own local data and with the ability to send and receive messages to and from other tasks, and shared memory programming model where all task/thread share a common address space to which they can read and write. The former is widely used in HPC domain and the latter in shared memory architectures (chip multi-processor and SMPs). The library specifications (e.g. MPI for distributed memory systems and Pthreads/OpenMP for shared memory systems) for implementing applications using these programming models also impacts performance as they provide the programmer with necessary abstractions for representing parallelism. Research is being carried out to address the shortcomings of current library specifications. For instance OpenMP is not well suited for handling irregular parallelism in applications and proposals like OpenMP tasking have been adopted to address this issue [6].

Furthermore, as indicated in the previous chapter we are going to make use of the SESC simulator and this provides a set of APIs that acts as a wrapper for the hosts Pthreads library. These APIs provided by SESC are only a subset of the Pthread library APIs. Since the application used OpenMP, the first task was to rewrite the applications to use Pthreads and then further modify it to use the SESC APIs as mentioned above. Also care was taken while porting, to ensure that the application only uses the available subset of SESC APIs.

This translation (OpenMP to Pthreads) motivated us to study the impact of threading libraries on application performance. The two implementations (one using OpenMP and the other using Pthreads) are compared to analyze the differences in scalability behavior when using different libraries . To carry out this analysis we use a shared memory system and the details of the evaluation platform are presented in Section 3.2.1.

3.2 Workload Evaluation

A concrete understanding of the application (workload) behavior is essential for identifying the various demands placed on the system architecture. Prior analysis of Minebench workloads has been restrictive on the number of threads used (analysis beyond eight threads has not been carried out) and the choice of threading libraries (only OpenMP) [36]. This evaluation addresses this by considering upto 32 threads for evaluation and by using different threading libraries (Pthreads/OpenMP). As a first step the application is profiled to identify its behavior. This involves identifying kernel sections, computing instruction mix and performing execution time profiling for the kernels. This is followed by scalability behavior analysis which explicates how the application behaves when it is scaled to run on multiple processors both using OpenMP and Pthreads. We start this section by first describing the evaluation platform, the metrics used for performing evaluation and some important issues with performance evaluation. After this we discuss about each application in detail in the following section.

3.2.1 Evaluation Platform

The hardware used for performing evaluation is a SGI Altix 4700 with a total of 128 Intel Itanium2 Montecito processors (at Barcelona Supercomputing Center). The SGI Altix machine is a distributed shared memory machine with a ccNUMA architecture. Each cpu has 2 cores working at 1.6Ghz supporting a 8MB L3 cache. It also had a 16KB instruction, data L1 cache and a 1MB instruction L2, 256KB data L2 cache per core respectively. The chapter presents the data obtained by running the applications (with upto 32 cores) with a view of understanding the computational phases and the overall behavior of the application when executed in a parallel environment. A detailed description of the architecture can be found in Appendix A. MPItrace [9] and OMPItrace [10] were used for source instrumentation, PAPI [43] for extracting performance counter values and Paraver [8] for visualization and analysis tool of parallel execution behavior. Although we needed to perform this scalability analysis on a multi-core processor, such a processor (with 32-cores) was not available at our disposal. So we had to in turn resort to performing this analysis on a SMP machine. Since both operate on the shared memory principle, we can assume the behavior of threads to remain the same.

3.2.2 Evaluation Metrics

The metrics utilized for measuring parallel performance are critical and they play a very important role in understanding application behavior. A lot of metrics have evolved over

the years to help determine the suitability of an architecture for efficiently executing a particular application. We make use of execution time, speedup and serial fractions to explain the execution behavior of applications on the architectural platform described in Section 3.2.1.

Speedup is a measure of how fast a parallel program is when compared to its serial counterpart. Speedup (S) for a machine with p processors/cores is defined by

$$S = \frac{T_s}{T_p}$$

where T_p is the parallel execution time (on p processors/cores) and T_s is the execution time obtained by running the application on a single processor/core.

Based on the measured speedup values, the serial fraction (S_{frac}) is derived. This metric was proposed in [27], to reason out the potential causes for scalability bottlenecks in applications. We use this metric to perform similar analysis for the applications that we are considering.

The serial fraction is computed as follows,

$$S_{frac} = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Intuitively, the serial fraction in any application should remain the same even if it scaled to many processors. This is because the serial fraction function assumes perfect load balancing and does not consider synchronization effects. Any irregularity in the derived serial fraction values indicate the influence of load-balancing effects, synchronization effects or architectural constraints on application scalability.

3.2.3 Evaluation Issues

The choice of the instrumentation method (source-based instrumentation, compile time instrumentation, etc.) and measurement technique (tracing, profiling, etc.) can have a large influence on the performance analysis results. We use source-based instrumentation method and it involves inserting manual instrumentation calls in the source code. The tool supports tracing which thoroughly captures and represents the occurrence of each event during the entire period of program execution. Since instrumentation can add lot of overhead to the measurement data (for instance, introducing a manual instrumentation event inside a loop body running for many iterations can have a significant influence on the application execution time), care has been taken inorder to avoid large perturbations. We accomplish this by first measuring the execution time without instrumenting the applications. Then we check if the overhead of instrumentation is under 5% . If this condition is satisfied we consider the experiment valid. In addition, we observed small variation in results when applications are executed multiple times, due to repeatability effects which arise when workloads are run on real machines, with real operating systems. The causes and effects of this variation are discussed in [40]. To compensate for this variance the results presented here are averaged out over multiple runs.

3.3 Workload Characterization Results

3.3.1 HOP

HOP [32] suggests a methodology for identifying groups of particles in N-body simulations. The application has various phases and the first phase involves assigning densities to each particle based on their neighbors. The application starts by constructing a KDtree for the particles and this is implemented by the `parakdBuildTree` kernel. After the tree construction step, `smSmooth` kernel is used to compute the density for each particle by considering Ndens nearest neighbours of the particle, implemented using a priority queue. This in turn makes use of `smBallSearch` and `smDensitySym` kernel to calculate densities for each particle. Then the particles are grouped/associated to the highest density neighbor. The `smBallGather` kernel traverses the priority queue list and assigns to each particle the address of the neighbour with the greatest density in a certain radius. This continues until the particle is its own densest neighbor and this scheme is assured to converge. To enhance the clustering quality, additional operations like merging and pruning are performed based on pre-defined threshold values and it also uses the `smBallGather` kernel. This phase determines the final group membership outcome for each particle.

Execution time profile and instruction mix Figure 3.1 (a) and 3.1(b) lists the execution time profile and the instruction mix configuration for the kernels in HOP. We can observe that `smSmooth` in the Density Computation Phase and `smBallGather` kernel in grouping and consume around 90%. of the overall execution time. Also from the instruction mix configuration chart, it is evident that the instruction mix constituting HOP is diverse. Unlike KMEANS and FUZZYMEANS (which are discussed below), the performance of the branch predictor is critical because this application does not have kernels that execute repeatedly.

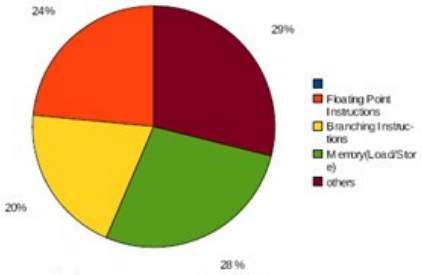
Scalability Analysis Figure 3.1 (c) depicts how HOP scales with increasing number of threads/cores. In this figure, each line has a label that indicates both the dataset size and the threading library used. For instance the line labeled 'Medium-omp' depicts the speedup as a function of the number of cores for the OpenMP implementation using medium sized dataset and the line labelled 'Medium-ptr' depicts the speedup for the Pthreads implementation using medium sized dataset.

The scalability analysis results presented here are obtained by running the application on the evaluation platform specified in Section 3.2.1. From the graph we can observe that the application scales until 4 threads, then marginally scales for up to 16 threads and does not scale beyond. Since the application is comprised of several kernels and each of them have varied behavior and we characterize them in detail to understand their impact on the overall scalability of the application.

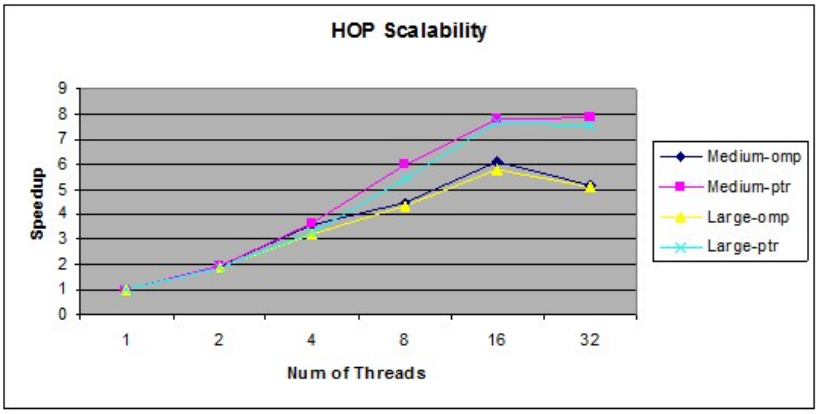
The scalability results of the individual kernels that make up the application are presented in Figure 3.1 (d). To better understand the impact of threading libraries,

Kernel	% excution time
prepareKDtree	2
smSmooth	41
smGather	50
smDensitySym	7

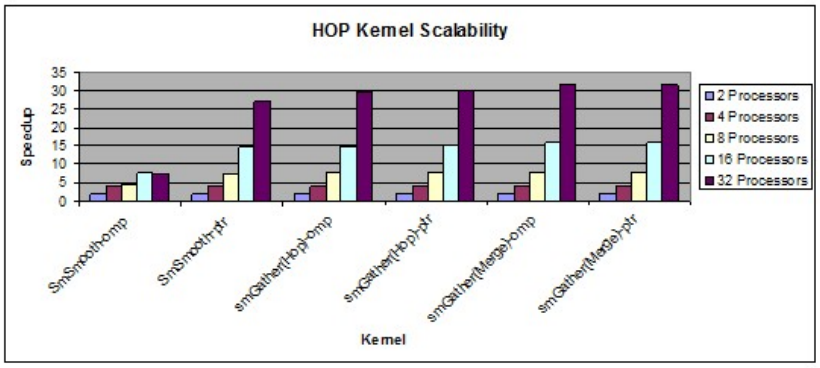
a) Execution time profile



b) Instruction Mix



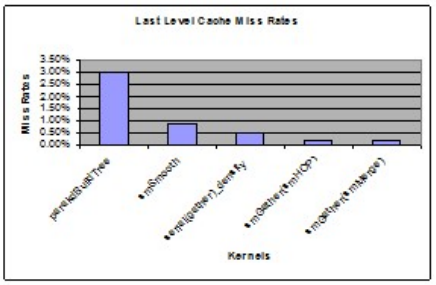
c) HOP Application Scalability



d) HOP Kernel Scalability

Threads/Procs	1	2	4	8	16	32
Speedup	1	1.96	3.62	5.96	7.8	7.89
Serial fraction		0.0200	0.0340	0.0480	0.0700	0.0900

e) Speed Up and Serial Fraction table



f) Last Level Cache Miss Rates

Kernel	INV/SNP Ratio
parakdBuildTree	0.22
smSmooth	0.0002
serial(gather)_density	0.0006
smGather(smHOP)	0.0001
smGather(smMerge)	0.0002

g) Cache Inv. Snoop Ratio

Figure 3.1: HOP Analysis.

we look at how individual kernels in the application behave and we specifically consider the 'Medium-ptr' and 'Medium-omp' to illustrate this. In Figure 3.1 (d) each bar indicates the scalability of the kernel when using particular number of processors. First, we compare the scalability results of the individual kernels to see if the use of different threading libraries can alter the scalability behaviour of the kernels.

From the graph (see Figure 3.1 (d)) we can observe that `smBAllGather` which makes up the largest kernel scales ideally even when parallelized with 32 threads (for Pthreads as well as OpenMP). We find that `smSmooth` kernel implemented in Pthreads scales well beyond 4 threads unlike with OpenMP where it only shows marginal scalability. The poor performance of OpenMP version is attributed to runtime overhead associated with handling significant amount of communication over large number of threads.

For `parakdBuildTree` kernel and the other serial kernel (not indicated in this figure) the use of Pthreads does not resolve any issue. `parakdBuildTree` does not scale well with more threads because of the excess overhead computations associated with parallel tree construction. The serial portion of the application also involves collecting the density values computed by individual threads in local array inorder to compute and update the global structure. The overhead involved with the serial function also keeps increasing with the number of threads.

Based on the scalability results, we compute the serial fraction values (for 'Medium-ptr') to help better understand the behaviour of the application and also reason out for the inconsistencies observed in speedup beyond 8 threads. The computed serial fraction values for HOP considering Pthreads implementation for medium dataset is shown in the Figure 3.1 (e). We can observe that the serial fraction is steadily increasing as the number of processor increases and this indicates the presence of parallel overhead. This overhead could be time spent on communication, synchronization or due to an architectural constraint. Next we perform analysis to identify the cause for this overhead.

We start by checking if the dataset size used for evaluation is leading to this scalability behavior. We compare the results with two different data sets (medium and large) to analyze the influence of the cache (see Figure 3.1 (c)). Since medium datasets have much smaller working set in memory when compared to lager datasets, they should have much lesser impact on performance (improved misrates). In this case however, we observe no noticeable difference in scalability for application using different dataset sizes which is an indication that the overheads observed not primarily due to cache capacity. Next, the miss rates for all these kernels on the last level of the cache hierarchy are derived and this is assumed to match memory bandwidth requirements, for checking if this is the cause of overhead. The last level miss rates have been computed from performance counter values which are used to monitor last level cache access. From the results in Figure 3.1 (f) we find that `parakdBuildTree` kernel has the highest miss rate (3 Misses for every hundred accesses). This is followed by `smSmooth` which has a miss rate of almost 1 for every hundred accesses. Also we monitor the invalidation transactions issued by a processor and the total snooping time to derive a ratio. The ratio computed for the

different kernels in the application are given in Figure 3.1 (g). From this data we can conclude that the kernels that make up majority of the application scales well. The scalability overhead is hence attributed to `parakdBuildTree` kernel and the other serial phases in the applications.

3.3.2 SCALPARC

SCALPARC [26] is used for formulation of decision trees in parallel from large datasets. In decision tree based classification all the attribute lists that are to be evaluated are first sorted once at the beginning. This is performed by distributing the number of lists equally among all the available threads in the `VRCompare` parallel kernel. Then a tree is constructed for class attributes based on other attributes in the dataset with a goal of having all the elements in a leaf belong to one class. The tree construction process involves two phases namely, the split determining phase and the splitting phase.

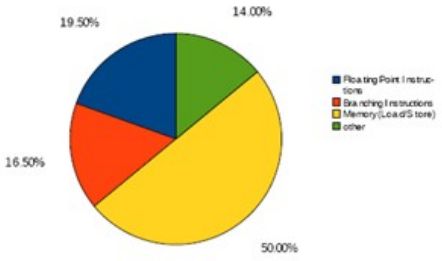
In the split determining phase each attribute is evaluated based on certain criteria (Giniindex) to determine the optimal split point. The `Giniindex` kernel used for this purpose is the most compute intensive and this is also implemented by distributing lists across the threads. This is followed by a serial phase which involves collecting the local optimal splitting information computed across the different threads and using them to determine the globally optimal split. Once the splitting point is identified the records have to split based on the decision. The splitting phase is constituted by two steps one which involves splitting split attribute lists and the other which involves splitting other non-split attribute lists. The process of splitting the split attribute and filling the hash table are carried out serially. This is again followed by a parallel phase which involves splitting the non-split attribute lists. The serial and parallel phases described above constitute the `Parclassify` kernel. Once the splitting process is finished we iterate again until we have all elements in each leaf node belonging to a specific class.

Execution time profile and instruction mix Figure 3.2 (a) and 3.2(b) lists the execution time profile and the instruction mix configuration for the kernels in SCALPARC. From the table it is clear that most of the computations are constituted by the three kernels (approximately 96 %). Also, it can seen that the around half of the instructions involve memory access either in form of loads/Stores. This indicates that the application is memory intensive.

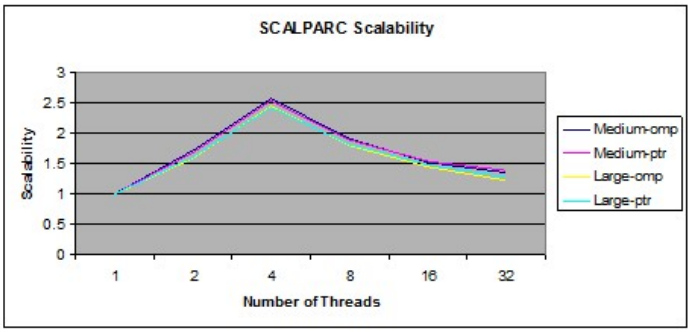
Scalability Analysis The graph in Figure 3.2 (c) shows application scalability behavior when executed with increasing number of threads. In this figure, each line has a label that indicates both the dataset size and the threading library used. For instance the line labeled 'Medium-omp' depicts the speedup as a function of the number of cores for the OpenMP implementation using medium sized dataset and the line labelled 'Medium-ptr' depicts the speedup for the Pthreads implementation using medium sized dataset. From the graph we can observe that the application scales marginally only until 4 threads, beyond which it does not scale. Scalability results of the kernels that

Kernel	% excution time
Gini_Index	39
parclassify	35
VRCompare	24

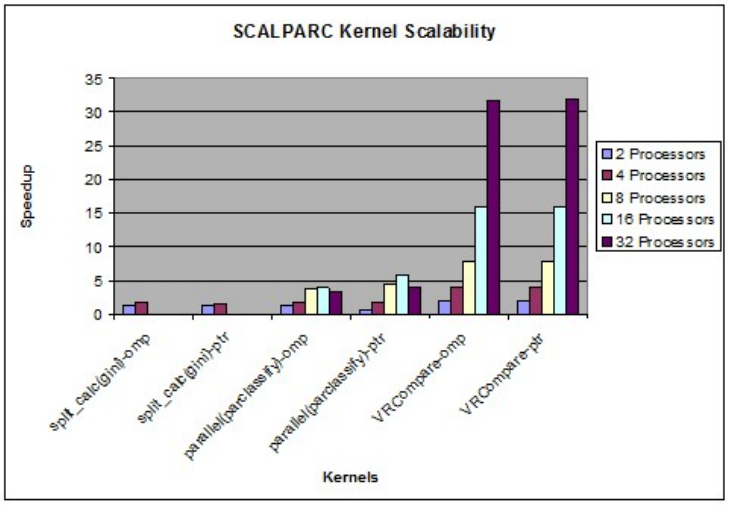
a) Execution time profile



b) Instruction Mix



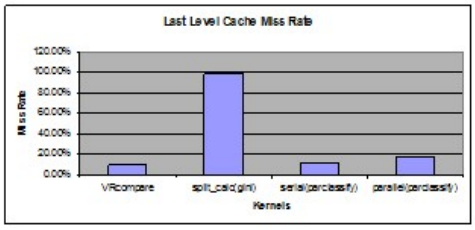
c) SCALPARC Scalability



d) SCALPARC Kernel Scalability

Threads/Proc	1	2	4	8	16	32
Speedup	1	1.6	2.44	1.8	1.45	1.23
Serial fraction		0.2500	0.2100	0.4900	0.6900	0.8000

e) Speed Up and Serial Fraction table



f) Last Level Cache Miss Rates

Kernel	INV/SNP Ratio
split_calc(gini)	0.0003
serial(parclassify)	0.2400
parallel(parclassify)	0.2200
VRcompare	0.0400

g) Cache Inv. Snoop Ratio

Figure 3.2: SCALPARC Analysis.

constitute the application are discussed.

The scalability results of the individual kernels are presented in Figure 3.2 (d). We look at how individual kernels in the application behave and we specifically consider the 'Medium-ptr' and 'Medium-omp' to illustrate the impact of threading libraries. In Figure 3.2 (d) each bar indicates the scalability of the kernel when using particular number of processors. First, we compare the scalability results of the individual kernels to see if the use of different threading libraries can alter the scalability behaviour.

From the graph (see Figure 3.2 (d)), we can observe that `VRCompare` kernel scales almost linearly and this is because each thread can sort the lists assigned to it in parallel independent of other threads. The parallel split determing phase, which uses the `Giniindex` kernel shows marginal scaling for upto four threads and does not scale beyond four threads. The serial portion of the application (splitting attribute lists) does not show any improvement because of additional threads. The part of the `parclasify` kernel that is responsible for nonsplit attribute lists scales marginally. From the graph in Figure 3.2 (d) we can observe that for most of the kernels the scalability behaviour is very similar upto a point (8 threads) beyond which Pthreads marginally outperforms OpenMP.

Based on the scalability results (for 'Medium-ptr'), we can observe that the serial fraction is steadily increasing as the number of processor increases and this indicates the presence of parallel overhead. The computed serial fraction values for SCALPARC considering Pthreads implementation for medium dataset is shown in the Figure 3.2 (e). This could be time spent on communication, synchronization or due to an architectural constraint.

We first analyze if the dataset size used for evaluation is leading to this scalability behaviour. Also, two different data sets (medium and large) are compared to observe the influence of the cache (see Figure 3.2 (c)). For SCALPARC we dont observe any noticeable difference in scalability because the working setsize is considerably larger than the last level cache size even for medium datasets.

The miss rates for all the kernels in SCALPARC on the last level of the cache hierarchy are computed (see Figure 3.2 (f)) . These results have been obtained with the help of performance counters which are used to monitor last level cache access and misses. From the results we find that SplitDetermining phase (which makes use of `GiniIndex` Kernel) has the highest miss rate (around 90 Misses for every hundred accesses). Also, the number of accesses made in the last level cache are considerably high. Because the size of the attribute list that we evaluate at each phase reduces in size as we proceed to the following level this miss rate value also improves (reduces to around 40 %. at the leaves). Although the miss rate at the last level is considerably high we can find that the kernel scales marginally for upto 4 threads. As memory bandwidth utilization details cannot be exactly obtained, one plausible reason for inverse scalability (increased execution time with increase in the number of threads) beyond 4 threads could be due to the architecture

of the SGI Altix4700 blade because it only has two sockets each capable of supporting a dual core Itanium connected to a local memory bank. Additional threads can fetch data from global address space only through the Interconnection network by accessing the remote memory banks. In addition, we monitor the invalidation transactions issued by a processor and the total snooping time to derive a ratio. We use this to analyse the influence of cache coherence and find that Splitting phase (`parclassify` kernel) has the largest ratio as shown in Figure 3.2 (g). The poor performance of this application can hence be attributed to the memory intensive nature of the kernels employed in the application.

3.3.3 KMEANS

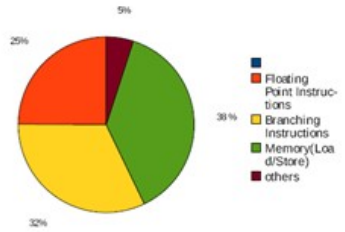
KMEANS [36] is among the most widely used algorithms for clustering datasets. Clustering is the process of grouping similar particles such that there is very high correlation among the different elements within a cluster and low correlation across the different clusters. KMEANS works by computing the cluster membership for each particle based on `Euclidean` distance. This is the most compute intensive kernel in the application and it is used to compute the distance between the element and the respective center. The algorithm first chooses random points as cluster centers. The next step involves associating each point to its closest center. `findnearest` kernel uses the computed distance values to find the closest center. `Cluster` Kernel makes up the serial part as it is involves collecting membership information from all the threads and performing a reduction operation on them to compute the new cluster centres. After this locally computed densities and center information is used to recalculate and identify new centers. This process is iterated until a user defined threshold is attained. `Euclidean` and `findnearest` together constitute the `nearestcenter` kernel phase and `Cluster` alone makes up the `updatecenter` kernel phase.

Execution time profile and instruction mix Figure 3.3 (a) and 3.3(b) lists the execution time profile and the instruction mix configuration for the kernels in KMEANS. From the table we can observe that the kernels constitute the major portion of the application (approximately 96 %). Although the instruction mix profile indicates a high percentage of branch instructions, the performance of the branch predictor performance overshadows this. It can be attributed to the fact that KMEANS repeatedly executes kernels which consist of loops that iterate over large indexes.

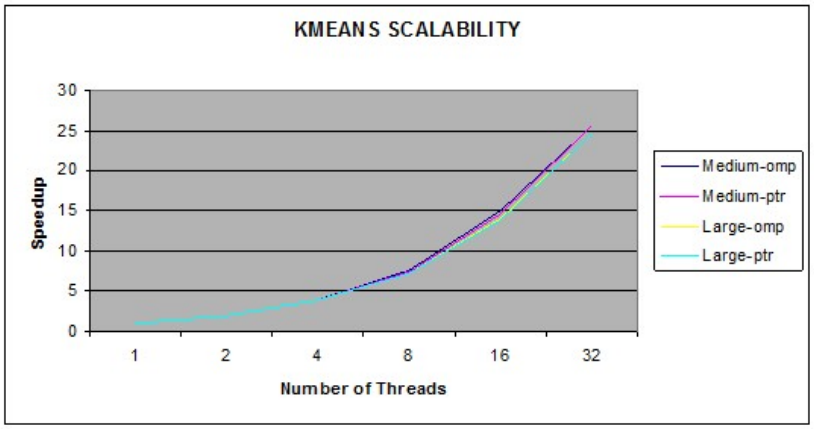
Scalability Analysis The scalability behavior of KMEANS is presented in Figure 3.3 (c). In this figure, each line has a label that indicates both the dataset size and the threading library used. For instance the line labeled 'Medium-omp' depicts the speedup as a function of the number of cores for the OpenMP implementation using medium sized dataset and the line labelled 'Medium-ptr' depicts the speedup for the Pthreads implementation using medium sized dataset. From the graph we can observe that the application scales reasonably for 32 processors.

Kernel	% excution time
Euclidean distance	70
find_nearestpoint	19
cluster	10

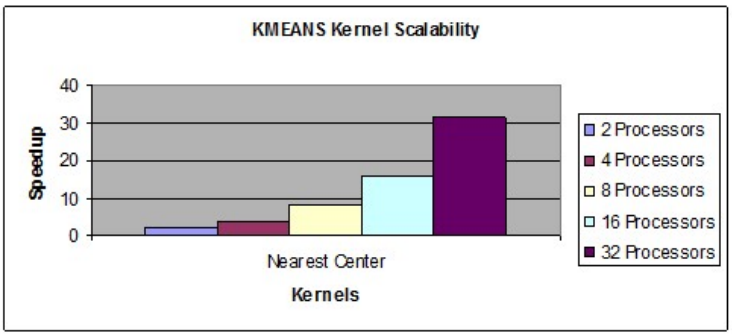
a) Execution time profile



b) Instruction Mix



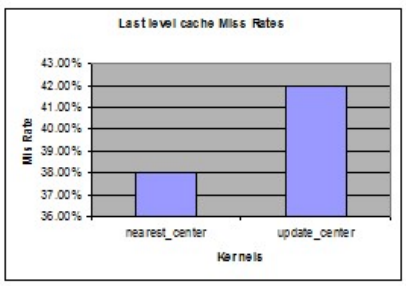
c) KMEANS Application Scalability



d) KMEANS Kernel Scalability

Threads/Procs	1	2	4	8	16	32
Speedup	1	1.9	3.92	7.44	14.68	25.47
Serial fraction		0.0520	0.0060	0.0100	0.0050	0.0080

e) Speed Up and Serial Fraction table



f) Last Level Cache Miss Rates

Kernel	INV/SNP Ratio
nearest_center	0.0003
update_center	0.0500

g) Cache Inv. Snoop Ratio

Figure 3.3: KMEANS Analysis.

Scalability results of the kernels that constitute the application are given in Figure 3.3 (d). To better understand the impact of threading libraries, we look at how individual kernels in the application behave and we specifically consider the 'Medium-ptr' and 'Medium-omp' to illustrate this. In Figure 3.3 (d) each bar indicates the scalability of the kernel when using particular number of processors. From the graphs we can observe that the `nearestcenter` phase shows speedup of around 30. This speedup is mainly because of the scalable nature of distance computation routine as indicated before. Once this phase is complete it is followed by a serial phase `updatecenter`, which is responsible for computing the cumulative center values and also updating the centers at the end of the current iteration. From the graph (see Figure 3.3 (d)), we can observe that for the kernels in this application, the scalability behaviour is very similar across Pthreads and OpenMP.

Based on the speedup values (for 'Medium-ptr'), we compute the serial fractions. We can observe that the serial fraction varies irregularly thereby suggesting load imbalance, communication or synchronization overheads. We first analyse the impact of working set size on the kernels to observe the influence of the large working set on the cache, as shown in Figure 3.3(e). We find that the larger dataset only causes marginal difference and this is because the larger dataset though computationally intensive (due to increase in vector size), has minor impact on the memory performance.

The miss rates observed for all the kernels in last level of the cache hierarchy are computed. These results (see Figure 3.3 (f)) have been obtained with the help of performance counters. From the results we find that `updatecenter` membership computation phase has the highest miss rate. Although the miss rate seems high the number of L3 accesses is less when compared to memory intensive algorithm like ScalParC. The invalidation to snooping ratio presented see Figure 3.3 (g)), indicates that the effect of cache coherence is minimal in all the kernels which suggests that the bottleneck cannot be due to coherence issues across the different threads.

3.3.4 FUZZYMEANS

In FUZZYMEANS clustering, each point has a degree of belonging to clusters, rather than belonging completely to just one cluster. It is an extension to KMEANS and is a statistically formalized method whereby a particle can belong to more than one cluster with a certain computed probability. The application starts off by first computing the sumdistances of each particle from all the centers and is done as a part of the `fuzzysum` kernel, which makes use of the `Euclideandistance` kernel for computing distance values. It is implemented by distributing the particles equally among all the threads. This sum is then used for computing the degree of membership for each particle and updating all the local centers accordingly. `Cluster` kernel is responsible for computing the degree of membership for each centre based on the results obtained from the previous kernel. The serial portion of the kernel performs reduction of the results across the various threads and derives new centre values. This clustering process is iterated until a specified threshold limit is attained.

Execution time profile and Instruction Mix Figure 3.4 (a) and 3.4(b) lists the execution time profile and the instruction mix configuration for the kernels in FUZZYMEANS. From the table we can observe that the kernels constitute the major portion of the application (approximately 96 %) as observed in KMEANS. The difference is that the `cluster` kernel in FUZZYMEANS is more time consuming than the `Euclideandistance` kernel owing to the complexities of computing multiple memberships for each particle. Also, we can notice that the application repeatedly executes kernels which consist of loops that iterate over large index values as observed in KMEANS. The parallel phase in the `cluster` kernel makes up `nearesrcenter` and the serial phase in the `cluster` is responsible for `updatecenter`.

Scalability Analysis The scalability behaviour of FUZZYMEANS is presented in Figure 3.4 (c). In this figure, each line has a label that indicates both the dataset size and the threading library used. For instance the line labelled 'Medium-omp' depicts the speedup as a function of the number of cores for the OpenMP implementation using medium sized dataset and the line labelled 'Medium-ptr' depicts the speedup for the Pthreads implementation using medium sized dataset. From the we can make an observation that FUZZYMEANS scales reasonably for 32 processors and shows a speedup of around 20.

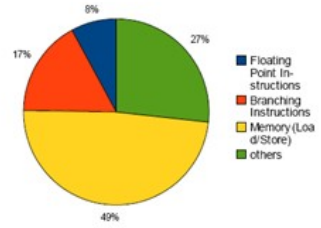
Scalability results of the kernels that constitute the application are given in Figure 3.4 (d). To better understand the impact of threading libraries, we look at how individual kernels in the application behave and we specifically consider the 'Medium-ptr' and 'Medium-omp' to illustrate this. In Figure 3.4 (d) each bar indicates the scalability of the kernel when using particular number of processors. From the graphs we can observe that the `sumfuzzy` kernel which inspite of performing distance computation (scalable operation), shows a speedup of around 25, the reasoning behind which will be discussed. It is followed by nearest cluster phase which uses distance information computed to identify the degree of association with each of the clusters and shows a speedup of roughly around 18. We compare the threading libraries to see if scalability behavior of the kernels is influenced. From the results (see Figure 3.4 (d)) we can observe that for the kernels in FUZZYMEANS the scalability behaviour is very similar across Pthreads and OpenMP.

Based on the speedup values we compute the serial fraction (see Figure 3.4 (e)). We can notice that the serial fraction varies irregularly and this suggests load imbalance, communication or synchronization overheads etc. Also, two different data sets (medium and large) are compared using the graph in Figure 3.4 (c). The difference in scalability was found to be marginal (as in KMEANS).

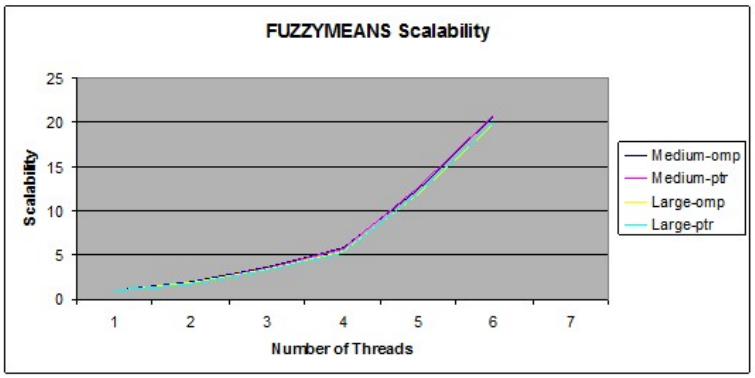
The last level cache miss rates and invalidation to snooping ratio values for the kernels in FUZZYMEANS are computed with the help of performance counters (see Figure 3.4 (f)). We draw a comparison across the computation phases in KMEANS and FUZZYMEANS with the help of these values. Firstly, the `sumfuzzy` kernel in FUZZYMEANS has a very similar memory access pattern to nearestcenter cluster in

Kernel	% excution time
Euclidean distance	39%
cluster	57%
find_sum	4%

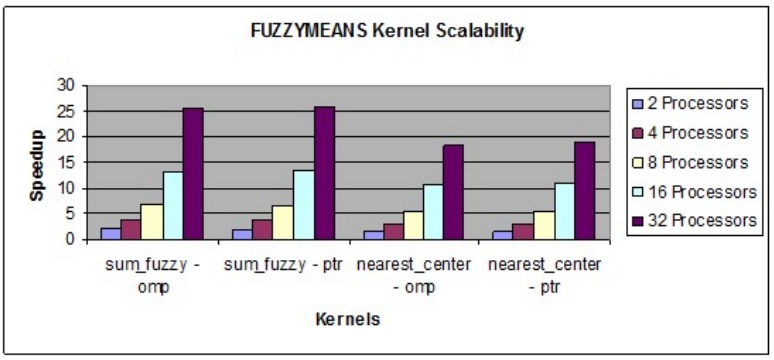
a) Execution time profile



d) Instruction Mix



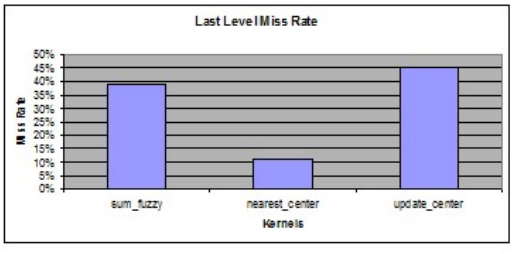
c) FUZZYMEANS Application Scalability



d) FUZZYMEANS Kernel Scalability

Threads/Procs	1	2	4	8	16	32
Speedup	1	1.7	3.32	5.8	11.4	21.3
Serial fraction		0.1700	0.0700	0.0480	0.0260	0.0160

e) Speed Up and Serial Fraction table



f) Last Level Cache Miss Rates

Kernel	INV/SNP Ratio
sum_fuzzy	0.015
nearest_center	0.0003
update_center	0.0600

g) Cache Inv. Snoop Ratio

Figure 3.4: FUZZYMEANS Analysis.

KMEANS and this is evident from the last level cache miss rates (both have similar values and almost equal number of accesses). We can also observe that the invalidation to snooping ratio, presented in Figure 3.4 (g) is larger for `sumfuzzy` and this is because the kernel updates a shared location after every call to distance routine, as opposed to nearestcuster kernel where it is updated once for each point. As for the nearestcluster kernel, the ratio remains the same and there is an improvement in the miss rate which can be atributed to temporal locality. `Updatecluster` kernel also has similar values when compared to KMEANS beacuse of the serial operations involved in updating cluster centers. Hence we conclude that the serial phase in FUZZYMEANS constitutes the bottleneck.

3.3.5 Conclusion

In this chapter we carry out performance analysis of applications to better understand the application behavior and also identify architectural/algorithmic bottlenecks that prevent the application from scaling to multiple threads. Based on performance measurement results in the previous section we have made few observations about the characteristics of data mining application. These are listed below.

1. Data mining applications exhibit data parallelism. Although this was observed in the benchmarks that we investigated, this cannot be generalized because there are applications that also exhibit task parallelism (depending on the algorithm parallelization technique).

2. The serial regions in these applications have varying computation complexities. For instance, some of the applications have serial phases that only perform the 'reduction' operation on the datasets, whereas for other applications, they comprise of large non-parallelizable regions of the kernels.

3. The applications exhibit differences in memory access, communication, synchronization behavior, computation intensity and scalability and this varies across the different kernels in the application as observed from the results presented in the previous sections.

4. The threading library used for parallelizing the application also provides some additional benefits for improving performance when the application is scaled to large number of threads. Although we observe this effect on the evaluation platform used for this study, we have to perform similar evaluations on other platforms to make a generic conclusion regarding the choice of threading library.

Scalable Multi-core Architectures for Data Mining Applications

4

In the previous chapter we presented our observations on the behavior of data mining applications from the Minebench benchmark suite. Since the diversity and the representativeness of the candidate benchmarks have already been discussed in Chapter 2, we can consider these benchmarks to be fairly representative of the domain. In this chapter, we use these observations to formulate architectural design decisions, which can help improve application scalability. First, in Section 4.1 we present a generic data mining workload model constructed using the observations made previously. Next in Section 4.2 we explore and identify the architecture alternatives that we will evaluate for scalable application execution. Section 4.3 describes the evaluation framework in detail, Section 4.4 presents the results obtained from the baseline homogeneous chip multi-processor that will be used as reference and Section 4.5 concludes the chapter. In Chapter 5, we will present the impact of these architectural alternatives in comparison to the baseline architecture.

4.1 Data Mining Workload Model

Based on the application characteristics observed in Chapter 3, we present a general model of data mining workload in Figure 4.1. This hypothetical workload has two parallel regions and three serial regions, which makes up the two kernels. The computation intensity at each phase is represented by the thickness of the line. This directly relates to the execution time for that particular phase.

The parallel regions can be categorized into two types. In the first type, tasks (work) are shared across the threads participating in the parallel region. This is generally occurs in loops where the iterations (work) are shared across the threads, either statically or dynamically. This region cannot have any form of synchronization operations (only an implied barrier at the end). In the second type, all the threads participating in the parallel execution, are assigned individual tasks (work). Unlike the former type, the threads in the latter communicate and synchronize among themselves to achieve coordination during task execution. Kernel 1 represents a parallel region of the first type and kernel 2 represents a parallel region of the second type. It should also be pointed out that all the parallel regions present in the workload express data parallelism. As a consequence the thickness of the lines in all the parallel regions are uniform.

The serial regions constitutes the phases, where only a single thread can exist. It is important to execute the serial phases as efficiently as possible because they play a significant role in the overall performance. The serial phase also have varying

computational intensities. For instance, Kernel 1 is computationally more intensive than the serial reduction phase.

KMEANS & FUZZYMEANS uses parallel region with implicit synchronization at the end, similar to that of kernel 1 and we term these as worksharing applications. HOP & SCALPARC uses parallel regions with explicit communication and synchronization and so its behavior is similar to that of kernel 2. We term these as non-worksharing applications. We use this workload as a representative for identifying plausible architectural alternatives for exploration and understanding the impact of each architectural alternative on application scalability.

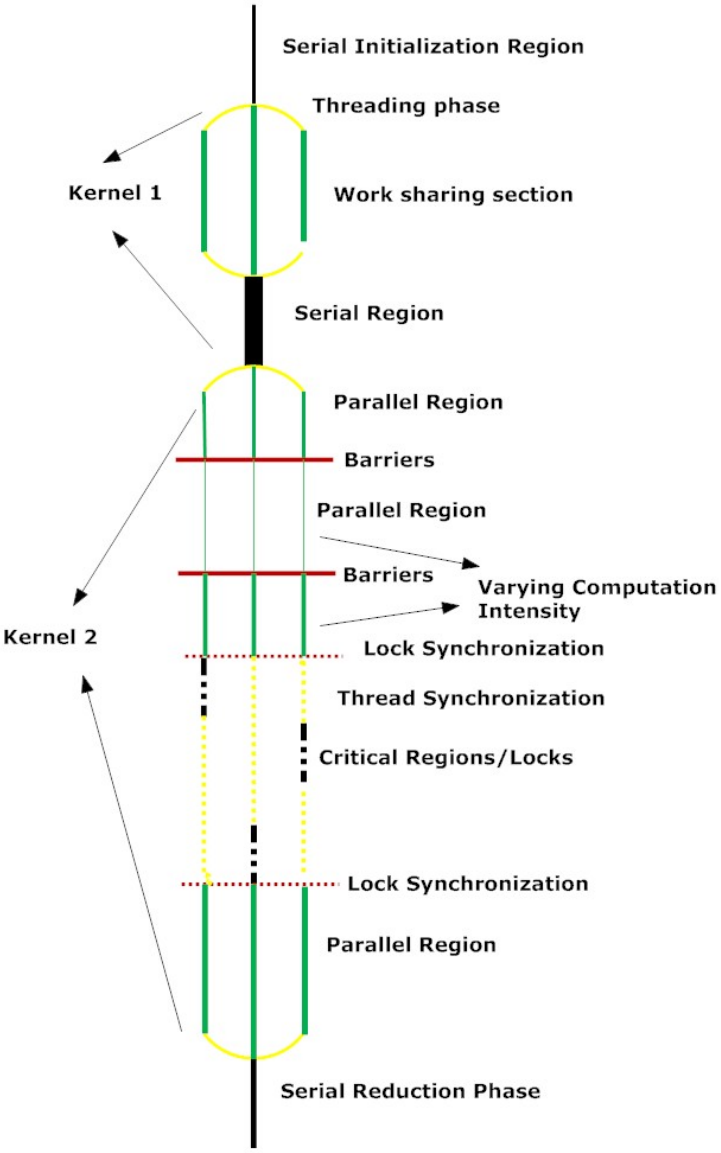


Figure 4.1: Data Mining Workload Model

4.2 Exploring Architectural Alternatives for Scalability

There are various possible avenues for exploring alternatives that help application scalability and we classify them into three distinct categories: system architecture and organization, memory system and communication and interconnection network. In this work we do not address the communication and the interconnection network aspects. Prior works in Memory subsystem characterization using data mining workloads have revealed several possible enhancements as indicated in chapter 1, that help improve memory performance. We rather focus our attention on the system architecture and organization aspects as this is least addressed.

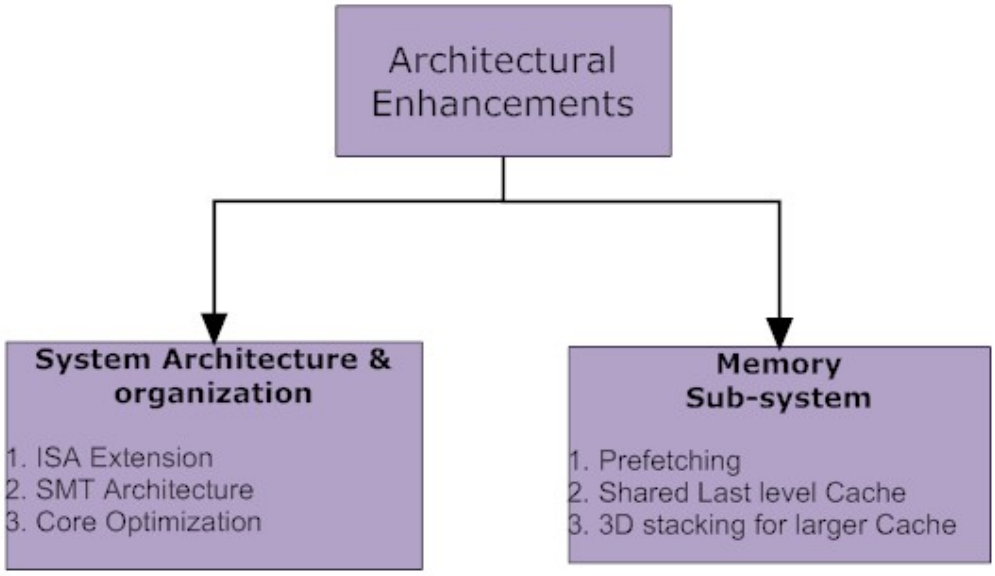


Figure 4.2: Architectural Enhancements Classification

The system architecture and organization avenue ranges from using techniques like SMT for hiding memory latencies, proposing ISA extensions for better speedups to careful core provisioning and multi-core heterogeneity management for improving overall system throughput. Figure 4.2 lists several possible architecture enhancements for data mining applications under system architecture and memory subsystem category. We restrict our analysis mainly to the system level aspects as it focuses on improving the applications computational throughput. Hence we investigate the following architectures.

4.2.1 Homogeneous Chip Multi-Processor

We first investigate the performance of these applications on a simulated homogeneous Chip Multi-Processor (CMP) architecture (see Figure 4.3 A). The size of the single baseline core used in this can either be small, mediocre or large based on the computational throughput requirements. Larger cores are preferred for exploiting coarse grain parallelism and smaller cores for fine grained parallelism. The main reason behind investigating homogeneous CMP can be attributed to the data parallel nature of the applications

(i.e. the computations that each thread performs remain the same, the difference lies in the data that each thread uses). Another reason is that these architectures represent the current design trend in the microprocessor industry. The performance results derived using homogeneous architecture for these applications on the simulator are discussed in Section 4.4 and will be used as a baseline for comparison.

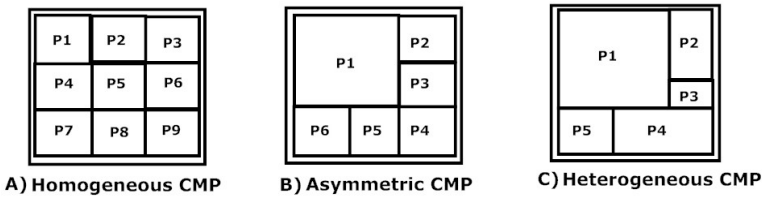


Figure 4.3: Scalable Multi-core Architectures for Data Mining Applications

4.2.2 Asymmetric Chip Multi-Processor

The performance of the modeled workload (as shown in Figure 4.1) can be enhanced by either accelerating the serial or the parallel computation phases or both. Large serial phases can easily prevent parallel applications from scaling to multiple threads (bottleneck) and hence addressing this becomes very crucial. Several architectures like Asymmetric Chip Multi-Processor (ACMP) and Accelerated Critical Sections (ACS) [41] have been proposed to address this issue efficiently. The basic idea behind ACMP/ACS is to have one large core that can efficiently handle the serial phases of the application and have several homogeneous cores that help in achieving high computational throughput during parallel execution phase (see Figure 4.3 B). ACS in addition executes the critical section computations on the larger core to yield better performance in comparison to ACMP. This requires operating system and architecture support as the threads executing the critical section have to be moved to the large core for execution.

As indicated in the observations presented in the previous chapter, data mining applications comprise of several serial phases each with varying computational complexities. This makes them an ideal candidate for asymmetric architectures. Also, the scheduling overhead associated with ACMP is minimal (could be simply based on priority). We investigate how data mining workloads behave on ACMP architecture and discuss the impact that these architectures have on the application scalability. The details pertaining to this and the results of these investigations are discussed in Chapter 5.

4.2.3 Heterogeneous Multi-Core Processor

Heterogeneous Multi-Core Processors (HMCP) comprise of cores each with different execution capability (see Figure 4.3 C). We consider a single ISA heterogeneous multi-core architecture (i.e., all cores have the same ISA) as the candidate for our evaluation. This choice of ISA is mainly limited by the capability of the simulator to simultaneously support multiple cores, each with different a ISA. The benefits of HMCP have been investigated in [28] and it has been found to offer superior performance

for exploiting both intra and inter thread diversity. This investigation was however carried out for multi-programmed workloads. In the previous chapter we observed that data mining applications exhibit varying behavior at different phases in the application and hence these workloads can be viewed as a candidate for HMCP architecture.

In Chapter 5, we also investigate the potential for heterogeneous multi-core for scalable execution of data mining applications. More importantly we compare the performance results of a naive scheduling policy (mapping thread to cores) to a resource aware scheduling policy and discuss their merits and drawbacks. Also we investigate the impact of these scheduling strategies on the overall scalability of the application.

4.3 Experimental Setup

We first simulate a 16-core homogeneous CMP, that is used as baseline for comparison. As discussed in Chapter 2, we make use of the SESC simulator for this study. First, we discuss the simulation timing and accuracy issues with the simulator. We limit simulation time by making appropriate selection of input datasets, reducing iteration count for iterative algorithms and fast-forwarding instructions (phases) in the applications which are not essential for each of the benchmarks. Although the influence of dataset size during execution on a real machine is minimal, it largely influences the simulation time, when using a simulator. One way to reduce simulation time is by using reduced input sets, which not only achieves reduced execution time but is also successful in replicating program behavior comparable to the reference input data set. The Minebench benchmark suite does not currently provide reduced input sets to address the simulation time issues. This leaves us with an option of selecting a particular dataset from the available datasets. Small datasets will cause variations if the working set can be held in cache, leading to effects like superlinear speedups. Large data sets on the other hand, can lead to exorbitant simulation times thereby influencing the extent to which investigation can be carried out. We use medium dataset to mitigate the effects of having a large/small datasets. For iterative algorithms we reduce the simulation time by not letting them iterate until the algorithm reaches convergence (we limit the number of iteration to reduce execution time). We also employ simulation fast-forwarding (emulation) wherever possible.

As for accuracy, a detailed study regarding the validity of the SESC simulator has already been carried out by Weaver et al. in [16]. We use the MIPS R12000 architecture configuration provided by them as the baseline core (single core) for modeling multi-core architectures, since this core configuration has already been used for prior validation studies (reduced modeling effort and improved modeling accuracy). We use a shared last level cache (L2 cache) to improve memory performance as shown by jaleel et al in [25] and MESI protocol for cache coherence in the simulated and . The table shows the configuration of the simulated architecture.

Processor Features	Configuration
Number of Cores	16
Core feature	300MHz R12000 out-of-order, 4-issue 33 arch registers 64 physical registers
Memory System	L1i: 32kB, 2-way, 64B L1d: 32kB, 2-way, 32B L2 : 4MB, 2-way, 128B(Shared)
Branch Predictor	2048 entry 2-bit
Coherence	MESI Protocol
Memory Bus	4:1 CPU/BUS ratio

Table 4.1: Baseline CMP Configuration.

4.4 Baseline CMP Scalability

We measure the scalability of these applications on a 16 core simulated homogeneous CMP. First we report performance of a single thread on the CMP and then scale it up until the number of thread contexts equals the number of cores in the architecture. We had to restrict our evaluation to 16 cores because anything beyond this core count led to several fold increase in simulation time. The scalability results are presented in Figure 4.3. We do not discuss the bottlenecks in detail here as these workloads have already been evaluated in chapter 3, although on a SMP machine. The main purpose of this evaluation is to have baseline performance figures that can be used for comparison.

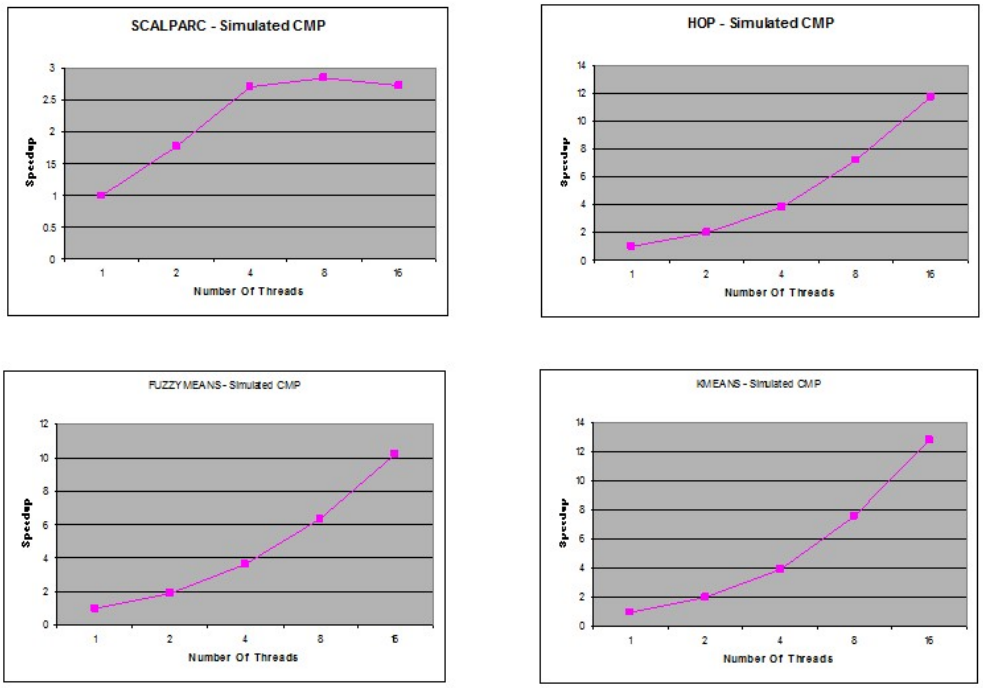


Figure 4.4: Simulated CMP Scalability

We list some key architectural differences between the two platforms that explains the difference in scalability behavior across the CMP and the SMP results (Chapter 3).

First, the communication and synchronization costs associated with CMP architectures are much lower when compared to a traditional SMP machine. Next, the available memory bandwidth, memory (cache) size per thread is much higher for SMP when compared to CMP architecture wherein the core/threads have to share the available memory bandwidth. Although there are other differences, the variation in application performance across the two platforms can mainly be attributed to these two aspects.

4.5 Conclusion

In this chapter, we first presented a generic model of the data mining workload constructed based on the characteristics identified in the previous chapter. Then we explored the possible architectural enhancements and identified suitable ones for further investigation. Next, we discussed the experimental issues involved and finally presented the results from the baseline CMP processor that is to be used as reference for comparison.

5 Evaluation and Results

This chapter evaluates the proposed architectural alternatives in order to identify the most scalable alternative for executing data mining workloads using the benchmarks described in the previous chapters. In Section 5.1, we look at the architectural specifications of the Asymmetric Chip Multi-Processor (ACMP) and Heterogeneous Multi-Core Processor (HMCP), described in the previous chapter. In Section 5.2 and Section 5.3, we analyze the performance of the ACMP and HMCP configurations, respectively.

5.1 Architectural Specifications

In addition to the baseline MIPS R12000 (R12K) configuration, we use two hypothetical processor configurations derived out of the R12K. We will refer to these configurations as R12K- and R12K+. The configuration details are listed below in Table 5.1.

	R12K-	R12K	R12K+
Core	300 MHz,In-order 2-Issue	300 MHz,Out-of-order 4-Issue	300 MHz,Out-of-order 6-Issue
Memory Hierarchy	L1:D-16K(2way) I-16K	L1:D-32K (2way) I-32K	L1:D-48K (2way) I-48K
Branch Predictor	1024 entry 2-bit	2048 entry 2-bit	g-share

Table 5.1: Core Configuration Table - R12K- ,R12K, R12K+.

We use these processor configurations to investigate the performance of the architectural alternatives considered and then compare the different architectural alternatives. It must be noted that all the architectural alternatives that will be investigated, assuming an equal area budget, unless specified otherwise. The cost model that we use for area and performance for each of the derived cores is simple and is similar to the model used in [22]. We use this to model the R12K+ and R12K- that are to be substituted in place of R12K cores. Under this model, when x small R12K/R12K- cores are replaced with one large R12K+/R12K core, the sequential performance of the large core improves by a factor between $x^1/2$ and $x^1/3$ over the small core.

5.1.1 ACMP Architecture Configuration

ACMP, as indicated in Section 4.2.2, uses one large core in order to speedup the serial phase in the application and several small/mediocre homogeneous cores to improve the computational throughput of the parallel phases in the application. We model an ACMP configuration, using two distinct cores, namely R12K and R12K+. Though the performance of ACMP varies with the configuration of the large core, we restrict our

evaluation to a single configuration (R12K+), which is sufficient to understand the benefits of ACMP. When using the baseline configuration, we employ 16 cores (threads) of the R12K, while in the ACMP configuration, we replace two R12K cores by one large R12K+ core, and hence, employ 15 cores (threads), though within the same computational area budget, as indicated in Table 5.2.

CMP(N Cores)	ACMP(N-1 Cores)	HMCP(N Cores)
16 R12K - baseline	15 (14 R12K, 1 R12K+)	16 (13 R12K, 2 R12K-,1 R12K+)

Table 5.2: CMP, ACMP and HMCP Configurations

5.1.2 HMCP Architecture Configuration

The HMCP architecture comprises of many cores, each with different execution capabilities. We model an HMCP configuration, using three distinct cores, namely R12K- (small), R12K (mediocre) and R12K+ (large), as specified in Table 5.1. Though the number of possible configurations for a given area budget is large, we restrict our analysis to a single configuration, where we are able to investigate the performance asymmetry introduced as a result of incorporating small cores in the design. We modify the ACMP configuration discussed in Section 5.1.1, by replacing a single R12K with two R12K- cores to achieve a heterogeneous configuration, as indicated in Table 5.2.

It must be noted, that though we restricted the architectural proposals to 16 cores, this configuration was adequate to investigate the scalability issues. We had to restrict the number of cores used for simulation because increasing the core count beyond 16 to 32,64 etc. led to several fold increase in simulation time.

5.2 R12K- vs R12K vs R12K+

First, we present the performance of the larger core (R12K+) and the smaller core (R12K-) in comparison to the mediocre core (R12K) for the serial regions in the application. This comparison provides an estimate of the execution time improvement that we can achieve by employing a large core and the difference in the serial region performance offered by the large and the small core.

In Figure 5.1, the bar labeled R12K+, depicts the serial region execution times on the R12K+ processor and the bar labeled R12K- depicts the serial region execution time on the R12K-. All the results are normalized to the R12K. From the graph we can observe that the large core performs better for all the applications. KMEANS showed the maximum benefit of around 35% from using the large core and this can be attributed to the simplicity of the computations in the serial 'update center kernel'. FUZZYMEANS also uses an 'update center kernel', but in addition has a threshold computation logic which is computationally intensive when compared to KMEANS and hence, shows an improvement of around 24%. The serial region in SCALPARC uses the values computed by the other threads to find the best splitting point and then performs

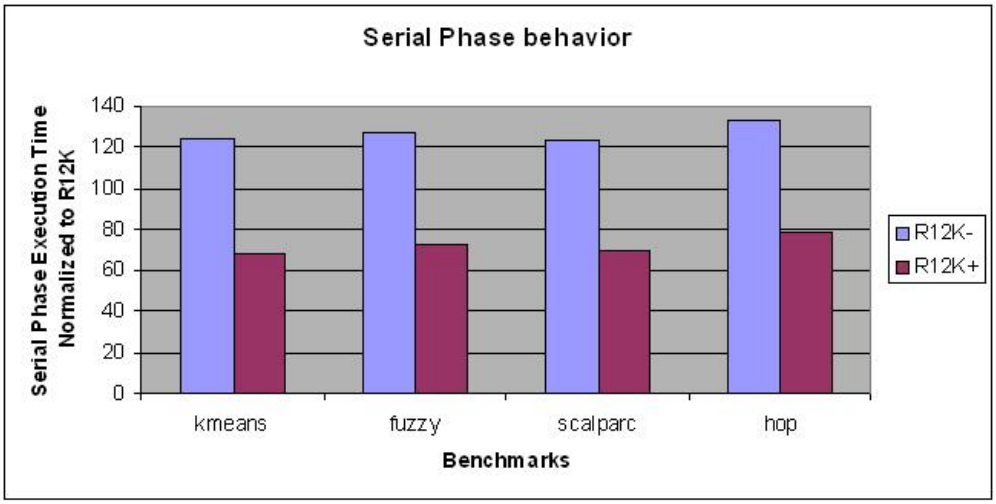


Figure 5.1: R12K+ vs R12K

simple spliiting operations based on the values computed and shows a 29% improvement. HOP shows the least improvement of 20% among all the benchmarks, since the serial region in HOP involves lot of memory accesses to update the densities for every particle. Similarly, using a smaller core to execute the serial regions negatively impacts performance as shown using R12K- configuration and this is because of the reasons presented above. Among the benchmarks, HOP shows the maximum slowdown using R12K- and this is due to the effect of reduced cache size on density update operations.

Next, we measure the performance of the benchmarks using a single thread. The first bar in the graph (see Figure 5.2) represents the normalized execution time on a R12K- core and the second bar in the graph represents the normalized execution time on a R12K+ core. From this graph, we can observe that these two configurations have varying impact on performance of the benchmarks. The difference in performance can be attributed to the architectural configuration of the cores used in the design as indicated in Table 5.1. SCALPARC shows the least difference across the two configurations and this is because the application is constrained by memory bandwidth bottleneck. Since equal memory bandwidth is assumed for all cores, having a larger core does not provide any additional benefit (for memory bandwidth). Moreover, the improvement in cache size and issue width only affects the performance marginally. On the other hand, HOP, KMEANS and FUZZYMEANS show good improvement in performance due to large cache size and improved issue width in the larger configuration.

5.3 ACMP Performance Analysis

In this section, we analyze the gains from using a larger core in the ACMP configuration, by comparing the performance of the applications on the ACMP configuration to that

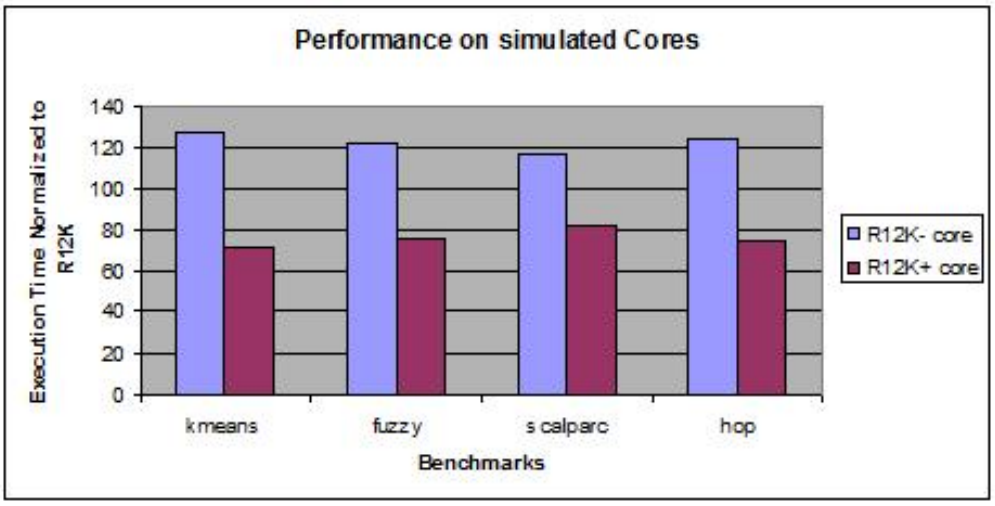


Figure 5.2: Single Thread Performance on R12K, R12K-, R12K+ cores

on the reference CMP configuration.

We investigate the performance of the ACMP proposal against the CMP (baseline) and analyze how the improvement in the serial region execution time, as discussed in Section 5.2, impacts performance scalability of the application. We perform this investigation on a 15-core ACMP configuration specified in Table 5.2, with 7 and 15 threads respectively (8 and 16 CMP equivalent threads). This provides insights into the application behavior when it scaled to multiple threads.

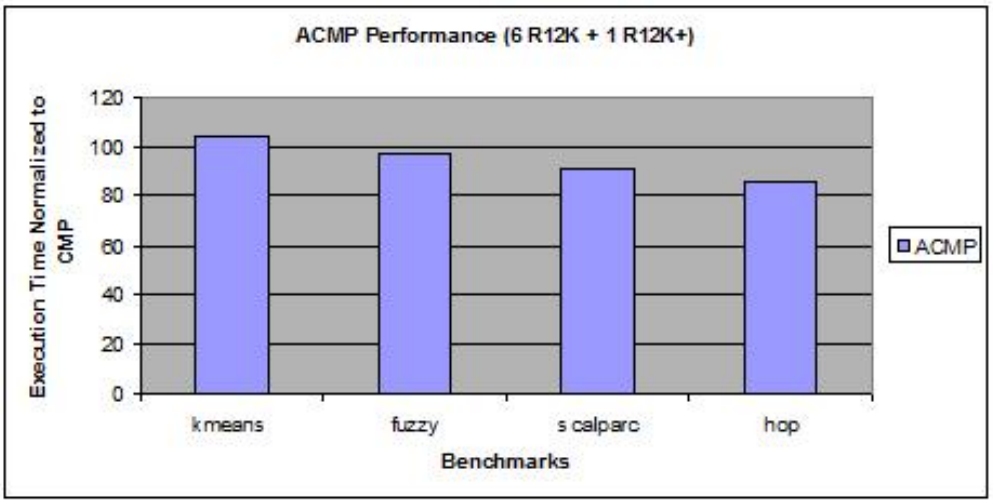


Figure 5.3: ACMP vs CMP (for 8 CMP equivalent threads)

The graph in Figure 5.3 shows the execution times for each application (when executing 7 threads mapped on ACMP - 6 threads on R12K and 1 thread on R12K+

core) on the ACMP architecture normalized to equivalent baseline CMP architecture (8 CMP threads on R12K cores), with an exception in the case of HOP, where the implementation only supports thread counts in powers of two. Hence for HOP, we use a slightly larger configuration for the ACMP (7 threads mapped on R12K and 1 on R12K+) and normalize the execution times to equal core count CMP. From Figure 5.2 we can observe that for KMEANS the baseline CMP configuration outperforms ACMP configuration by 6% because the serial portion only constitutes a small portion of the application. For such throughput intensive applications, the performance improvement gained by the additional thread in baseline CMP outweighs the enhancement in the serial portion brought about by ACMP. For FUZZYMEANS, the serial portion is computationally intensive when compared to KMEANS and the benefit of having a faster core (ACMP) for computing the serial phase helps in marginally outperforming the CMP execution time by 3%. SCALPARC on the other hand shows very little improvement beyond 4 threads and does not scale beyond 8 due to memory bandwidth limitations. As the parallel region execution time does not show any significant improvement, improving the serial region execution time helps improve performance thus resulting in a 9% improvement. For HOP however, the large core helps improve performance by reducing the miss rates in the serial phases. This coupled with the performance benefit gained by using an additional thread (8 threads instead of 7 threads) results in an improvement of around 15%).

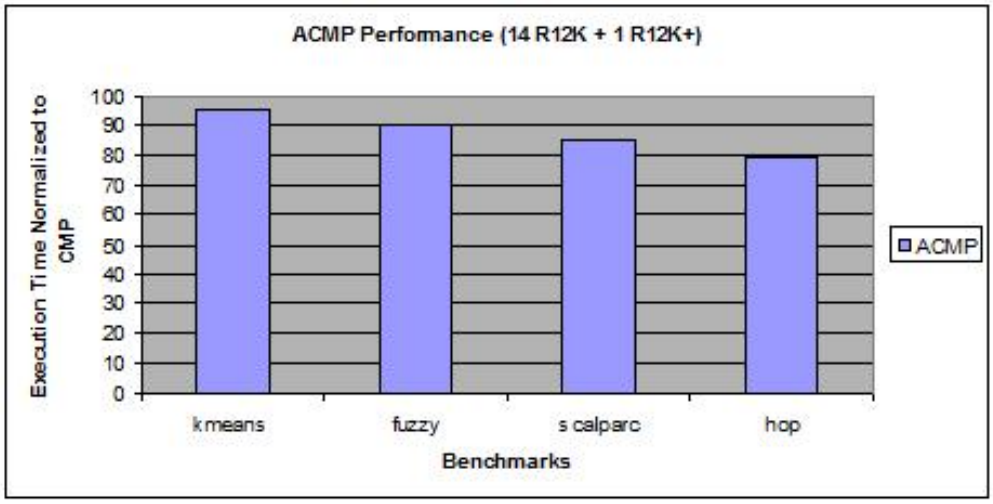


Figure 5.4: ACMP vs CMP (for 16 CMP equivalent threads)

The graph in Figure 5.4 depicts the impact of ACMP architecture on application performance when parallelized using 15 threads in a ACMP configuration (14 threads on R12K and 1 thread on R12K+ core threads). This experiment was performed to analyze the impact of ACMP when the application is scaled beyond eight threads. For all these applications we have observed that the computations involved in the serial region increases with the increase in the number of threads. For instance in KMEANS,

as the number of threads increases to 16, the computations in the serial phase also increases. This causes the execution time of KMEANS on ACMP to outperform an equal area CMP, in spite of being throughput oriented as specified previously. Although the improvement over CMP is marginal at around 4%, these results clearly indicate growth in serial computation phases with increase in the number of threads. For FUZZYMEANS however, the increase in the number of threads does not have a large impact (around 10%) on performance in spite of the growing serial complexity. This is mainly because the computational intensive phase in the serial region (threshold computation) remains the same. The increase in the number of threads impacts the computations for calculating the 'updated center values' and this translates only to a marginal improvement in speedup. We observed similar trends in performance for SCALPARC and HOP, wherein a performance improvement of 13% was observed for the fomer and 21% for the latter.

From the graphs, we can conclude that the inclusion of a larger core in the design is a good architectural alternative to improve scalability of data mining applications. Now, we theoretically derive the speedup bounds for ACMP and baseline CMP configurations used previously, using Amdahl's Law to see if the simulated ACMP performance improvement matches the theoretical improvement. The speedups for ACMP and CMP are calculated using the equations given below where p_{frac} denotes the parallel portion of the application, s_{accel} represents the improvement in the serial execution time because of using the large core and n represents the number of cores. We can notice from the speedup equations that ACMP has one processor less than baseline CMP and this is because ACMP uses one R12K+ core in place of two R12K cores.

$$S_{CMP} = \frac{1}{1 - p_{frac} + \frac{p_{frac}}{n}}$$

$$S_{ACMP} = \frac{1}{\frac{1-p_{frac}}{s_{accel}} + \frac{p_{frac}}{n-1}}$$

The graph (see Figure 5.5) shows the speedup of applications on the ACMP architecture derived using Amdahl's Laws in comparison to the CMP architecture, for $n = 16$. The graph indicates that the theoretical speedup values for ACMP is lesser than CMP whereas the simulation result indicates otherwise. This difference in theoretical and observed speedup values can be attributed to assumptions made my Amdahl's law. First, the law assumes that the serial fraction is independent of the number of threads, but for benchmarks we are considering we have pointed out that it grows with the number of threads. Next it assumes that the parallelization is free (increasing the number of threads leads to linear increase in speedup of the parallel region) and does not consider additional overheads like load balancing effects, architectural resource constraints etc. These assumptions are the potential cause for the difference in theoretical and simulated performance.

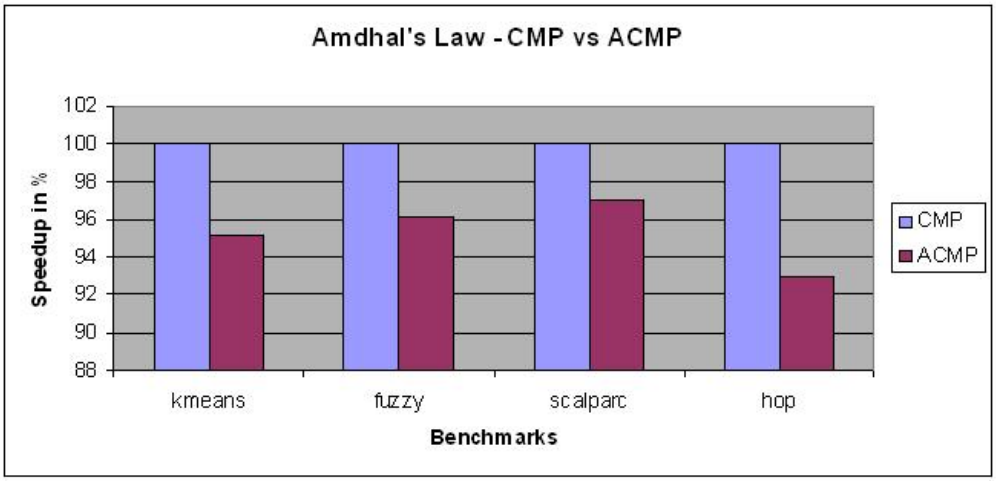


Figure 5.5: Amdahl's Law - ACMP vs CMP (for 16 CMP equivalent threads)

5.4 HMCP Performance Analysis

In this section, we analyze the impact of using a smaller (slower) core in the HMCP configuration, by comparing the performance of the benchmark applications on HMCP configuration against the baseline CMP configuration.

We perform this investigation for the configuration specified in Table 5.2 with 8 and 16 threads (CMP equivalent), to understand the impact of heterogeneity on scalability. As pointed out in the previous chapter, heterogeneous multi-core architecture brings in the overhead of scheduling threads to the appropriate core. Since the HMCP architecture that we model comprises of three different cores there are various possible ways by which threads could be scheduled across all the available cores. First, we carry out experiments under the assumption that the thread scheduler is unaware of the resource heterogeneity and it simply assigns the cores to the threads based on availability (naive scheduling policy). Next, we use a resource aware worksharing policy for scheduling work to threads and perform similar experiments. The analysis is carried out using all the applications (worksharing as well as non-worksharing) and the results obtained are compared to the baseline CMP results. Figure 5.6 shows the normalized execution time for each application on the HMCP architecture assuming a naive scheduling policy and using 8 threads.

Each bar has a label with two configurations. The first configuration represents the core that executes the serial phases in the application and the second represents the slowest core which executes the parallel phase. For instance, the first bar has the label R12K-, R12K-. This implies that the serial portion in the application is executed on the R12K- (small core) and the slowest core which executes the parallel region is also R12K-. When we compare the performance of baseline CMP and HMCP we can observe that the baseline CMP outperforms HMCP architecture for all the applications. Also,

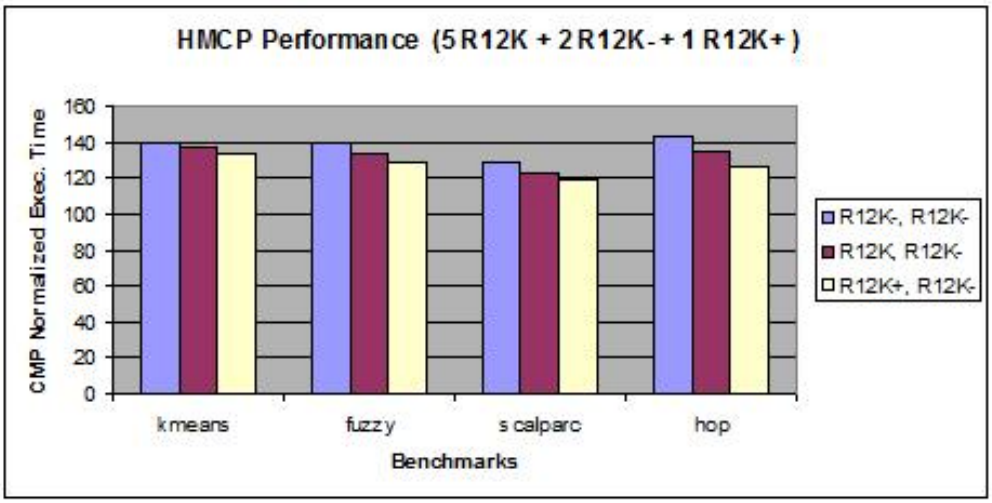


Figure 5.6: HMCP vs CMP (for 8 CMP equivalent threads)

we have to consider that the only difference between ACMP and HMCP configuration is the presence of two R12K- cores in place of a single R12K core. Hence this difference in performance (across CMP, ACMP and HMCP) could be attributed to the inclusion of these two slow cores in the HMCP configuration. We investigate the influence of having a small core in both non-worksharing as well as worksharing applications.

To better understand this performance slowdown in worksharing and non-worksharing workloads, we consider the application workload model represented in Chapter 4 (Figure 4.1). In this model at the end of parallel region there is a implicit barrier for worksharing apps and an explicit barrier for non-worksharing applications. This barrier ensures that the processing continues only after all the threads have reached this point in the control flow. Because the slow core offers least performance it reaches the barrier last and all the other threads a have to wait for this slow thread to reach the barrier point so that they can continue; this in spite of the fact that they have finished executing the tasks assigned to them. This explains the HMCP performance slowdown in comparison to the other architectural alternatives.

In addition to performance slowdown effects discussed previously, we can also observe the effects of performance asymmetry on a HMCP configuration. This effect arises because the serial regions in the application can be mapped to any of the available cores. In case the serial region is mapped to larger core it accelerates the serial region execution time (like ACMP) thereby improving application performance. On the contrary, if the serial region is mapped to the slowest core, this would adversely affect the application performance. This explains the difference in performance [7] across the different mapping schedules as indicated by the three labels in Figure 5.6, using a naive scheduling strategy. The impact of performance asymmetry on application performance has been studied in [7].

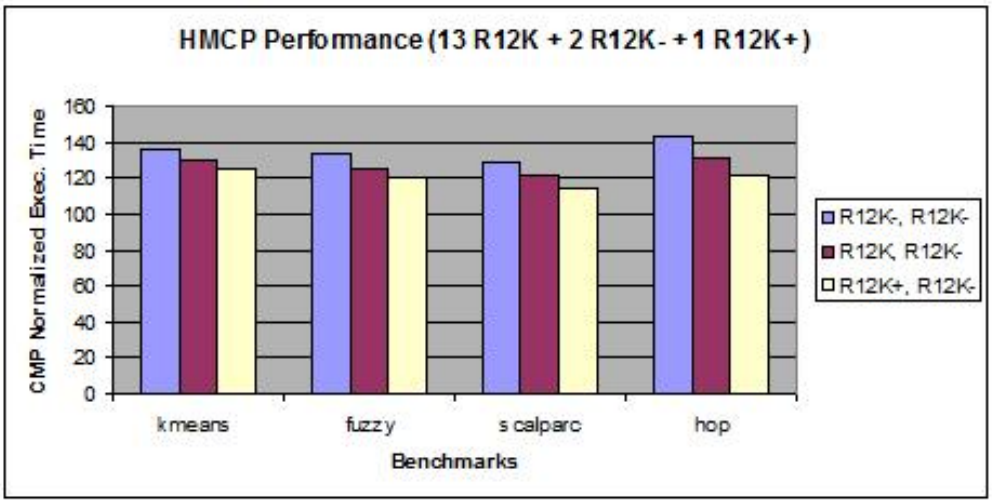


Figure 5.7: HMCP vs CMP (for 16 CMP equivalent threads)

Figure 5.7 shows the normalized execution time for each application when parallelized using 16 threads (similar to 8 threads as seen in Figure 5.6) on the HMCP architecture assuming a naive scheduling policy. This experiment was performed to analyze the impact of HMCP when the application is scaled up to 16 threads. From the results we can observe that the increase in the number of threads is leading to more asymmetry and this is reflected from the results presented in the graph. From the graphs (for 8 and 16 threads) we can observe that the performance asymmetry effects (marked by difference in execution time across the different possible scheduling combinations) are larger at 16 threads than at eight threads. This was constantly observed across all the applications.

To address this performance asymmetry, we use a resource-aware workload scheduling strategy, that allocates work to cores based on the execution capability of each core (performance that a core can deliver for a particular application). In other words, this scheduling strategy statically allots to the different cores, proportional share of the work that is made available to the system, based on the performance capabilities of those cores. In order to implement such a resource-aware scheduling, the cores are statically assigned scheduling weights (priorities) based on their computational area and and work is appropriately distributed across the cores using their scheduling weight information. For instance, let us consider a system configuration with 3 cores with computational area of 'x' and 1 core with that of '2x', the first 3 cores are allotted a weight of 20% each, while the 4th core is allotted a weight of 40%. This scheme ensures that, 40% of the work is scheduled to the 4th core, while 20% of the work is scheduled to each of the other 3 cores. This method assures proportional fairness in scheduling inorder to maximize throughput.

The results obtained by such a scheduling strategy is presented in Figure 5.8. We

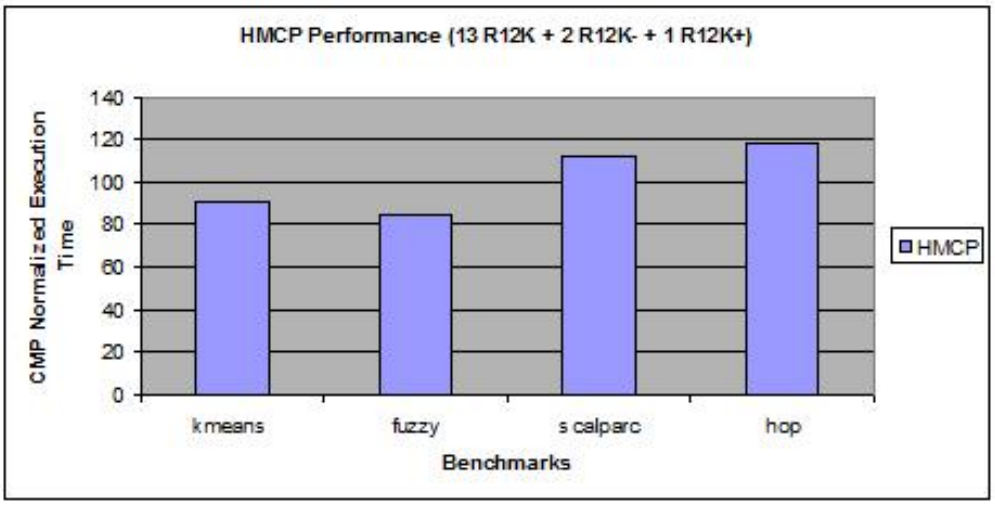


Figure 5.8: HMCP vs CMP (for 16 CMP equivalent threads with resource aware scheduling)

observe the impact of this scheduling strategy on worksharing as well as non-worksharing applications. The results indicate that worksharing applications perform much better on HMCP architecture in comparison to CMP architectures if resource aware scheduling strategy is used. This is observed in KMEANS and FUZZYMEANS. As for non worksharing applications, we see that they only show marginal improvement in performance when adopting a resource-aware schedule. The baseline CMP still outperforms HMCP architectures with resource aware scheduling for non-worksharing applications. This is because in case of worksharing applications (KMEANS & FUZZYMEANS) it is required to partition the work involved in iterations equally across the different threads. Our technique ensures that the larger core gets more work and the smaller core gets less work (iterations) thereby ensuring that the larger core waits for a lesser time at the implied barrier for the smaller core to finish execution. In case of non-worksharing applications however, there can be explicit communication at the any point in the parallel region. This causes us to loose the benefit of having a larger core because the thread on the smaller core always determines the critical path. This can be observed from the results obtained using HOP & SCALPARC as shown in the graph.

5.5 Conclusion

From the simulation results, we observed that ACMP architectures provided better performance scalability when compared to homogeneous configurations with similar computational area budgets. We also observed, that HMCP architectures with naive scheduling strategy offered lesser performance scalability when compared to both ACMP and baseline homogeneous architectures. However, when resource-aware work scheduling strategy was employed, for worksharing workloads, better performance scalability was observed. In case of non-worksharing workloads, the performance of ACMP and CMP clearly out-

performed the HMCP strategy. From the results we can arrive at a conclusion that ACMP architectures consistently perform better than HMCP architectures.

6 Conclusions and Future Work

In Section 6.1 we summarize the work that has been presented and draw conclusions from the results we have obtained. In Section 6.2 we make recommendations for future work.

6.1 Conclusions

The goal of this thesis was to investigate architecture alternatives that offer maximum scalability for the majority of the applications in a given domain. For this purpose, the data mining application domain was selected and specific architectural proposals were suggested based on the characteristics exhibited by applications in that domain. This work involved selecting candidate benchmarks, understanding architectural simulation platforms, investigating application scalability issues, identifying generic application characteristics, proposing alternatives to improve workload scalability on the basis of these characteristics and finally investigating the benefits of each of the architectural alternatives.

We choose the Minebench benchmark suite from the data mining application domain and we specifically worked with clustering and classification benchmarks namely HOP, SCALPARC, Kmeans and FUZZYMEANS. We then investigated various simulators to see if they match our simulation requirements. Based on the results of our evaluation we chose the SESC simulator for performing architectural evaluation. These applications were then ported to SESC, by translating the application source code (written in OpenMP) to use Pthreads and then further modified to use the SESC API's (subset of Pthread API'S).

To investigate the impact of threading library on scalability, we evaluated the Pthreads and the OpenMP versions of the application by running them on a SMP (SGI Altix) machine. Based on the experiments, we determined that for communication intensive tasks, the Pthread library was more scalable than OpenMP. The analysis of these workloads also helped in identifying characteristics that were common across the applications considered. We used these characteristics to identify architectural proposals for improving scalability during execution. Based on these characteristics, we identified that Asymmetric Chip Multi-Processor (ACMP) and Heterogeneous Multi-Core Processor (HMCP) architectures would be appropriate for these workloads and we then used the simulator to model them and also analyze the performance gains offered by each of them.

We observed that for all the applications, ACMP architectures provided better performance scalability when compared to homogeneous configurations with similar

computational area budgets. We also observed, that HMCP architectures with naive scheduling strategy offered lesser performance scalability when compared to both ACMP and baseline homogeneous architectures. However, when resource-aware work scheduling strategy was employed, for worksharing workloads, better performance scalability was observed. In case of non-worksharing workloads, the performance of ACMP and CMP clearly outperformed the HMCP strategy. From the analysis of the data mining benchmarks carried out on the different architectural alternatives, we can conclude that ACMP architectures consistently offer better performance scalability than HMCP architectures.

6.2 Future Work

There are several possible ways for extending this work. One way could be to incorporate detailed area, power and thermal models and use these merits to explore diverse architecture configurations. Another way could be to use additional benchmarks from this domain to identify common characteristics and use them to design and investigate specialized accelerator cores. Another possible extension could be, to incorporate detailed on-chip interconnection models in the simulation, to analyze the impact of communication delays on application performance and scalability. Studying the influence of Operating System on application scalability is another interesting subject.

Bibliography

[1] *Sgi*, 2007, `http://techpubs.sgi.com`.

[2] *Eetimes*, 2008, `http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=206901564`.

[3] *Kryder's law*, 2009, `http://www.scientificamerican.com/article.cfm?id=kryders-law`.

[4] *Top500*, 2009, `http://www.top500.org/system/details/9707`.

[5] Todd Austin, *Simple scalar hackers guide*, 2009, `http://www.simplescalar.com/docs/hack_guide_v2.pdf`.

[6] Eduard Ayguad, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang, *The design of openmp tasks*, IEEE Transactions on Parallel and Distributed Systems **20** (2009), no. 3, 404–418.

[7] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai, *The impact of performance asymmetry in emerging multicore architectures*, SIGARCH Comput. Archit. News **33** (2005), no. 2, 506–517.

[8] BSC., *Paraver reference manual*, May 1999.

[9] BSC., *Mpitrace users guide*, November 2000.

[10] BSC., *Ompitrace users guide*, November 2000.

[11] Gregory Buehrer, *Scalable mining on emerging architectures*, Ph.D. thesis, Columbus, OH, USA, 2008, Adviser-Parthasarathy, Srinivasan.

[12] Y. K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, M. Smelyanskiy, and M. Smelyanskiy, *Convergence of recognition, mining, and synthesis workloads and its implications*, Proceedings of the IEEE **96** (2008), no. 5, 790–807.

[13] Yu Chen, Wenlong Li, Junmin Lin, Aamer Jaleel, and Zhizhong Tang, *Data sharing analysis of emerging parallel media mining workloads*, HiPC, 2008, pp. 87–96.

[14] J. Pisharath et al., *Accelerating data mining workloads: Current approaches and future challenges in system architecture design*, Proc. 9th Int'l Workshop on High Performance and Distributed Mining, in conjunction with 6th Int'l SIAM Conf. Data Mining, 2006.

[15] Mark Hempstead et al., *Navigo: An early-stage model to study power-constrained architectures and specialization*, Modeling, Benchmarking, and Simulation(MOBS 2009), June 2009.

[16] Weaver et al., *Are cycle accurate simulations a waste of time?*, Workshop on Duplicating Debunking and Debunking (2008).

[17] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony D. Nguyen, Yen-Kuang Chen, and Pradeep Dubey, *A characterization of data mining workloads on a modern processor*, DaMoN, 2005.

[18] Processor Amol Ghoting, Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen kuang Chen, and Pradeep Dubey, *Cache-conscious frequent pattern mining on a modern*, In Proceedings of the International Conference on Very Large Data Bases (VLDB, 2005, pp. 577–588.

[19] Chris Gottbrath, *Overcoming the challenges of developing applications for the cell processor*, Linux J. **2008** (2008), no. 175, 4.

[20] Tom R. Halfhill, *RISC fights back with the Mips R12000 — SGI tweaks its chip with better bus bandwidth, wider parallelism, and stronger emphasis on FP performance*, Byte Magazine **23** (1998), no. 1, 49–??

[21] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk, *Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture*, SIGMETRICS Performance Evaluation Review (2004).

[22] Mark D. Hill and Michael R. Marty, *Amdahl's law in the multicore era*, Computer **41** (2008), no. 7, 33–38.

[23] Peter Hofstee, *Heterogeneous computing, hardware and software fundamentals, the cell broadband engine and its applications*, 2008, `http://ce.et.tudelft.nl/cecoll/?id=237`.

[24] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez, *Core fusion: accommodating software diversity in chip multiprocessors*, Proc. 34th International Symposium on Computer Architecture (34th ISCA'07) (San Diego, California, USA), ACM SIGARCH, June 2007, pp. 186–197.

[25] Aamer Jaleel, Matthew Mattina, and Bruce L. Jacob, *Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads*, HPCA, 2006, pp. 88–98.

[26] Mahesh V. Joshi, George Karypis, and Vipin Kumar, *Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets*, IPPS/SPDP, 1998, pp. 573–579.

[27] Alan H. Karp and Horace P. Flatt, *Measuring parallel processor performance*, Commun. ACM **33** (1990), no. 5, 539–543.

[28] Rakesh Kumar, *Holistic design for multi-core architectures*, Ph.D. thesis, UNIVERSITY OF CALIFORNIA, SAN DIEGO, 2006.

[29] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen, *Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction*, MICRO, ACM/IEEE, 2003, pp. 81–92.

[30] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen, *Conjoined-core chip multiprocessing*, MICRO, IEEE Computer Society, 2004, pp. 195–206.

[31] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas, *Single-ISA heterogeneous multi-core architectures for multithreaded workload performance*, ACM SIGARCH Computer Architecture News **32** (2004), no. 2, 64–64.

[32] Ying Liu, Wei-keng Liao, and Alok Choudhary, *Design and evaluation of a parallel hop clustering algorithm for cosmological simulation*, IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing (Washington, DC, USA), IEEE Computer Society, 2003, p. 82.1.

[33] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hllberg, Johan Hgberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner, *Simics: A full system simulation platform*, Computer **35** (2002), no. 2, 50–58.

[34] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, *Multifacet's general execution-driven multiprocessor simulator (gems) toolset*, SIGARCH Comput. Archit. News **33** (2005), no. 4, 92–99.

[35] Tomer Y. Morad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguad?, *Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors*, IEEE Computer Architecture Letters **5** (2006), no. 1.

[36] Berkin Özisikyilmaz, Ramanathan Narayanan, Joseph Zambreno, Gokhan Memik, and Alok N. Choudhary, *An architectural characterization study of data mining and bioinformatics workloads*, IISWC, 2006, pp. 61–70.

[37] Vijay Pai, Parthasarathy Ranganathan, and Sarita V. Adve, *The impact of instruction-level parallelism on multiprocessor performance and simulation methodology*, In 3 rd International Symposium on High Performance Computer Architecture, 1997, pp. 72–83.

[38] Paul Sack, *Sesc hackers guide*, 2009, `iacoma.cs.uiuc.edu/~paulsack/sescdoc/`.

[39] Kelly Shaw, *Understanding the working sets of data mining applications*, Eleventh Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-11), 2008.

[40] D. Skinner and W. Kramer, *Understanding the causes of performance variability in hpc workloads*, IEEE Workload Characterization Symposium **0** (2005), 137–149.

[41] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt, *Accelerating critical section execution with asymmetric multi-core architectures*, ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ACM, 2009, pp. 253–264.

[42] University of Rochester Computer Science Department, *Mint tutorial and user manual*, May 1993.

[43] UTK., *Papi users guide*, May 1999.

[44] Joshua J. Yi and David J. Lilja, *Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations*, IEEE Trans. Computers **55** (2006), no. 3, 268–280.

SGI Altix Architecture

A

The SGI Altix machine is installed in a 42U SGI rack each comprising of 4 Individual Rack Units (IRU) and each of these IRU's cab support ten compute/memory and I/O modules called blades. Each of these blades have ASICS, processors, memory and I/O chipsets and is mounted on a mechanical carrier. The system architecture for the Altix 4700 system is a fourth-generation NUMAflex DSM architecture known as NUMAlink 4. In the NUMAlink 4 architecture, all processors and memory are tied together into a single logical system with special crossbar switches (routers). This combination of processors, memory, and crossbar switches constitute the interconnect fabric called NUMAlink. There are four router switches in each 10U IRU enclosure. See Figure A.1.

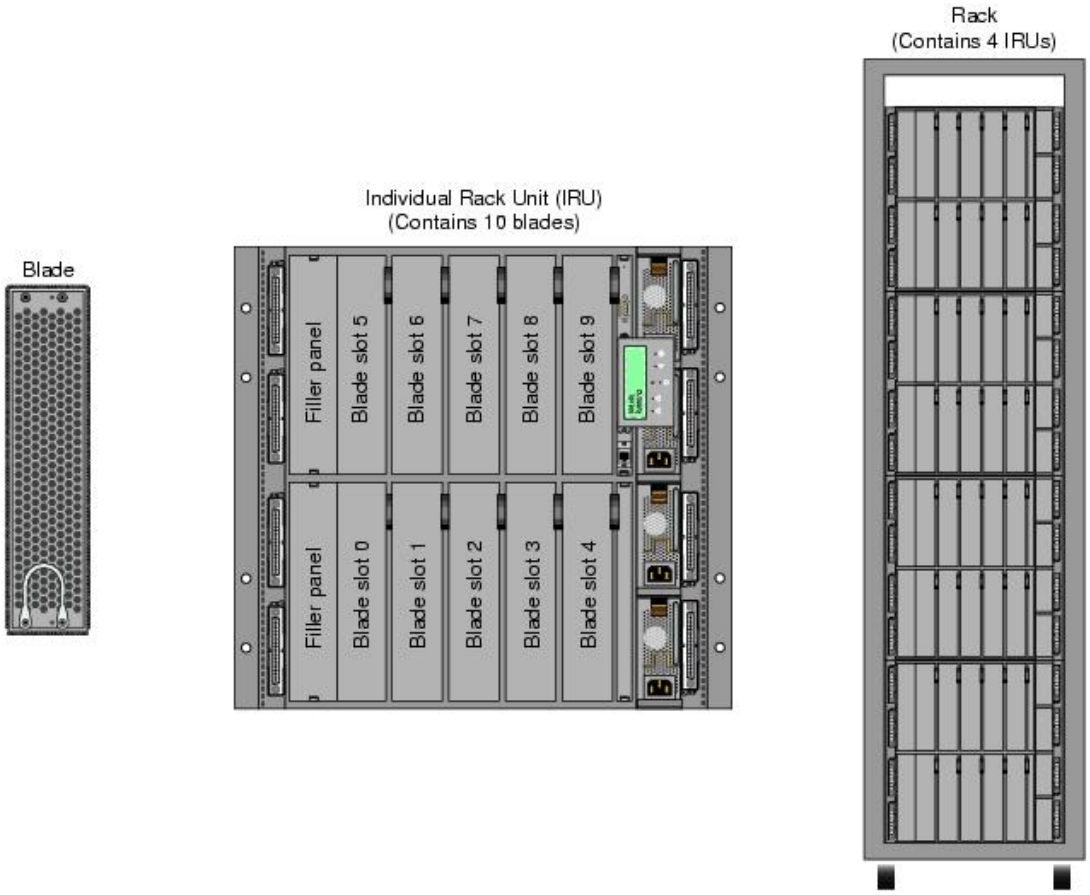


Figure A.1: 42U SGI Rack (taken from [1])

The basic expansion building block for the NUMAlink interconnect is the processor node; each processor node consists of a Super-Hub (SHub) ASIC and one or two 64-bit processors with three levels of on-chip secondary caches. The Intel 64-bit processors are connected to the SHub ASIC via a single high-speed front side bus. This specialized ASIC acts as a crossbar between the processors, local SDRAM memory, and the network interface. The SHub ASIC memory interface enables any processor in the system to access the memory of all processors in the system. Figure A.2 shows a conceptual block diagram of the blade's memory, compute and I/O pathways.

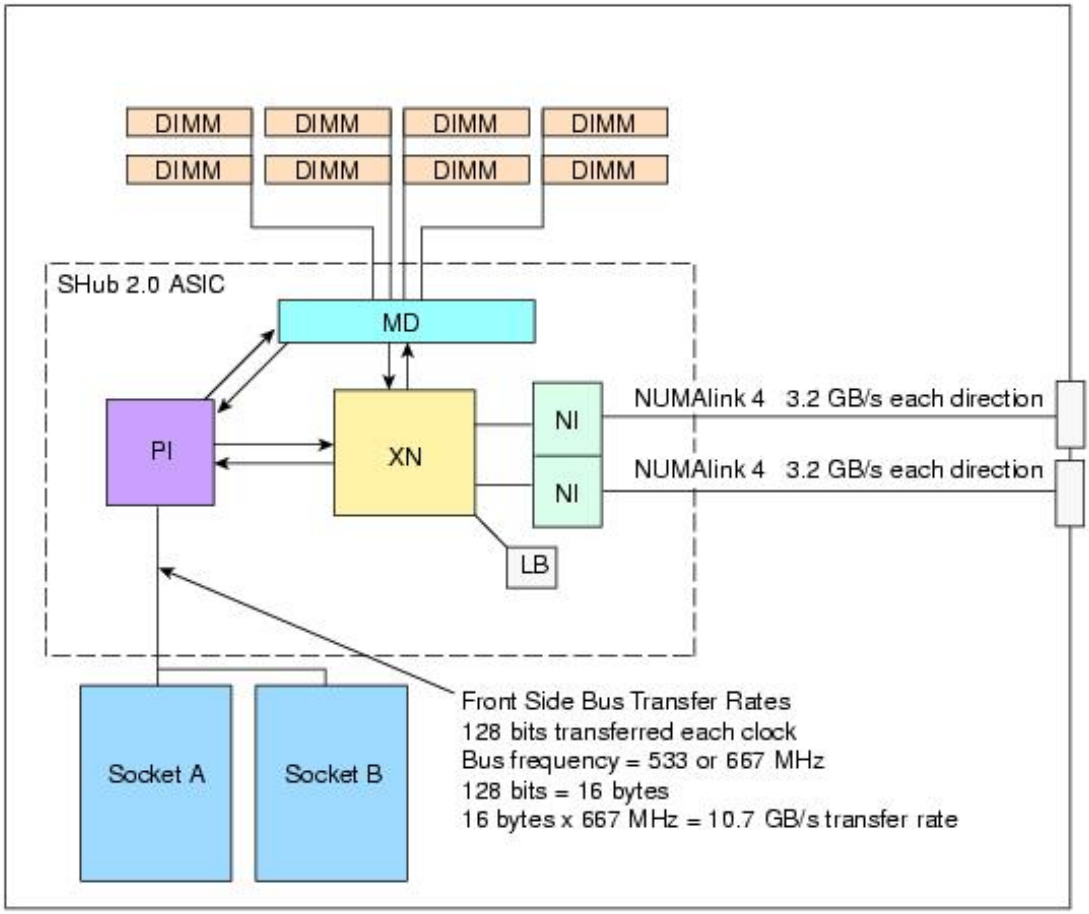


Figure A.2: SGI Blade Architecture (taken from [1])

Another component of the NUMAlink 4 architecture is the router ASIC. The router ASIC is a custom designed 8-port crossbar ASIC. Using the router ASICs with a highly specialized backplane or NUMAlink 4 cables provides a high-bandwidth, extremely low-latency interconnect between all processor, I/O, and other option blades within the system. In the Altix 4700 series server, memory is physically distributed both within and among the IRU enclosures (compute/memory/I/O blades); however, it is accessible to and shared by all NUMAlinked devices within the single-system image. This is shown

in Figure A.3.

The following are the sub-types of memory within a system:

- If a processor accesses memory that is connected to the same SHub ASIC on a compute node blade, the memory is referred to as the node's local memory and memory latency is lowest for these accesses.
- If processors access memory located in other blade nodes within the IRU, (or other NUMAlinked IRUs) the memory is referred to as remote memory.
- The total memory within the NUMAlinked system is referred to as global memory.

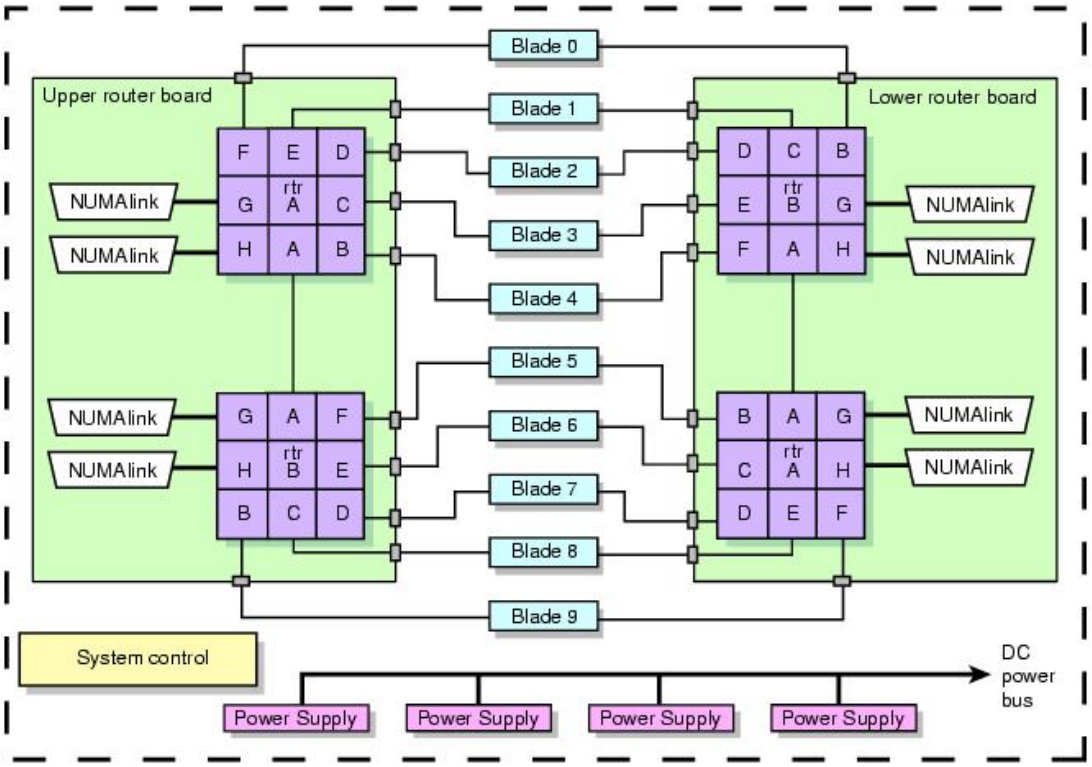


Figure A.3: SGI NUMAflex DSM architecture (taken from [1])

In DSM systems, memory is physically located at various distances from the processors. As a result, memory access times (latencies) are different or "non-uniform." The Altix 4700 server series use caches to reduce memory latency and this leads to the cache coherency issues. Although data exists in local or remote memory, copies of the data can also exist in various processor caches throughout the system and cache coherency is required to keep the cached copies consistent. This architecture uses a directory-based coherence protocol, where each block of memory (128 bytes) has an entry in a table that is referred to as a directory. Like the blocks of memory that they represent, the directories are distributed among the compute/memory blade nodes. A block of memory is also

referred to as a cache line. Each directory entry indicates the state of the memory block that it represents. For example, when the block is not cached, it is in an unowned state. When only one processor has a copy of the memory block, it is in an exclusive state. And when more than one processor has a copy of the block, it is in a shared state; a bit vector indicates which caches contain a copy. When a processor modifies a block of data, the processors that have the same block of data in their caches must be notified of the modification. The Altix uses an invalidation method to maintain cache coherence. The invalidation method purges all unmodified copies of the block of data, and the processor that wants to modify the block receives exclusive ownership of the block. The overview of the system configuration is given below.

- 128 cpus Dual Core Montecito(IA-64).
- Each one of the 256 cores works at 1,6 GHz, with a 8MB L3 cache and 533 MHz Bus.
- 2.5 TB RAM.
- 2 internal SAS disks of 146 GB at 15000 RPMs
- 12 external SAS disks of 300 GB at 10000 RPMS