İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

MODEL BASED PARALLELIZATION OF OBJECT ORIENTED SOFTWARE FOR MULTICORE SYSTEMS

Ph.D. Thesis By
Tolga OVATMAN

Department : **Computer Engineering**

Programme : **Computer Engineering**

JUNE 2011

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

MODEL BASED PARALLELIZATION OF OBJECT ORIENTED SOFTWARE FOR MULTICORE SYSTEMS

Ph.D. Thesis by
Tolga OVATMAN
(504052505)

Date of submission : **20 April 2011**
Date of defence examination : **16 June 2011**

Supervisor(Chairman) : **Assist.Prof.Dr. Feza BUZLUCA (ITU)**
Members of the Examining Committee : **Prof.Dr. Nadia ERDOĞAN (ITU)**
Prof.Dr. Emre HARMANCI (ITU)
Assoc.Prof.Dr. Can ÖZTURAN (BU)
Assist.Prof.Dr. Elif KARSLIGİL (YTU)

JUNE 2011

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

ÇOK ÇEKİRDEKLİ SİSTEMLER İÇİN NESNEYE DAYALI YAZILIMLARIN MODEL TABANLI PARALELLEŞTİRİLMESİ

DOKTORA TEZİ

Tolga OVATMAN
(504052505)

Tezin Enstitüye Verildiği Tarih : **20 Nisan 2011**
Tezin Savunulduğu Tarih : **16 Haziran 2011**

Tez Danışmanı : **Yrd.Doç.Dr. Feza BUZLUCA (İTÜ)**
Diğer Jüri Üyeleri : **Prof.Dr. Nadia ERDOĞAN (İTÜ)**
Prof.Dr. Emre HARMANCI (İTÜ)
Doç.Dr. Can ÖZTURAN (BÜ)
Yrd.Doç.Dr. Elif KARSLIGİL (YTÜ)

HAZİRAN 2011

“Our freedom to doubt was born out of a struggle against authority
in the early days of science. It was a very deep and strong struggle:
permit us to question - to doubt - to not be sure.
I think that it is important that we do not forget this struggle
and thus perhaps lose what we have gained.”

Richard Phillips Feynman, "The Value of Science",
address to the National Academy of Sciences (Autumn 1955)

FOREWORD

The paper you are about to read is not only a detailed summary of my doctoral study, but also the epilogue to my 30 years as a student. During the six years of my PhD education there are many people contributed to the studies that forms the basis of this thesis. I would proudly mention those people, of whom you may find the traces whilst reading the paper.

Dr. Feza Buzluca is the first person I would like to thank for his efforts as an advisor during my PhD studies. He is the reason of almost every bit of organized and properly prepared product that I was able produce during the last six years. Not only he influenced me as a researcher but also he was a role model for teaching and literature. I can't thank him enough for all the guidance and friendship through those six years.

Next, I would like to thank Prof.Dr.Nadia Erdoğan and Dr.Elif Karslıgil for supporting me with their knowledge and their advising effort during my thesis studies. Prof.Erdoğan has lead me with her experience and wisdom and Dr.Karslıgil always encouraged me with her energy and vision. They are very important assets of this thesis. I would also like to thank Prof.Dr.Emre Harmancı and Dr.Can Özturan for their valuable feedback and insightful remarks on my thesis studies. Prof.Harmancı was kind enough to spend several hours with me working on the manuscript and has helped me to improve it in a very authentic and unique way.

During my thesis study I was very fortunate to study with important researchers in Europe and United States. I would like to thank Prof.Thomas Weigert from Missouri University of Science and Technology for his contributions to my work on dependency patterns. I would also like to thank Prof.Michael Reichhardt Hansen and my colleague Aske Wiid Brekling from Technical University of Denmark for their collaboration during the early stages of my PhD study. Finally I would like to thank my colleague Gül Nildem Demir for her contributions on clustering for dependency patterns and also for her continuous support as a friend.

I would like to thank all my professors at Istanbul Technical University Computer Engineering Department, but especially Prof.Dr.Emre Harmancı and Prof.Dr.Eşref Adalı for their guidance and influences during the evolution of my scientific thinking and working discipline. I would also like to thank Dr.Cüneyd Tantuğ and Dr.Mustafa Kamaşak for their mental support.

I also owe many thanks to all my research assitant colleagues from İTÜ. I would like to thank Dr.Yusuf Yaslan, Dr.Burak Kantarcı, Dr.Mehmet Tahir Sandıkkaya and Dr.Ender Yüksel for their friendship, help and many contributions they have made during my studies. I would also like to especially thank Dr.Berk Canberk for walking together with me in almost all the phases of my PhD study.

I was not only supported academically during my PhD study; I wouldn't be able to complete my work without mental support of my friends. I want to thank Kenan Kule, Kıvanç Kaya and Cenk Çiçen for remembering me of the life that grows outside the lab whenever I need.

Last but not least I would like to thank my mother and father for influencing me with their infinite vision through all my life. I wouldn’t be here if hey haven’t encouraged me to continue as a graduate student and support me all the way. I would also like to thank Irfan Sakin for paving the way to my academic life as a role model with his engineering career.

“To be whole is to be part...” says Ursula K. Le Guin in her famous novel. I believe my PhD study(and myself as a PhD candidate) became a whole since all the people I mentioned above(and the ones I forgot the mention) became a part with their contributions. Le Guin continues as “...true voyage is return.”

June 2011

Tolga OVATMAN

Computer Engineer, M.Sc.

TABLE OF CONTENTS

Page

ABBREVIATIONS

API	**:**	Application Programming Interface
AST	**:**	Abstract Syntax Tree
CAWS	**:**	Cache-Aware Scheduler
CFS	**:**	Completely Fair Scheduler
CIID	**:**	Cumulation of Inner Island Dependencies
CI*policy*	**:**	Confidence Interval of a running time for a specific scheduling *policy*.
CMP	**:**	Chip Multi-processing
CPU	**:**	Central Processing Unit
CUS	**:**	Cache-Unaware Scheduler
GHz	**:**	Giga Hertz
GoF	**:**	Gang of Four
GPU	**:**	Graphics Processing Unit
IC	**:**	Integrated Circuit
JNI	**:**	Jaca Native Interface
JOMP	**:**	An OpenMP-like Interface for Java
Ln Cache	**:**	Level-n Cache
NCD	**:**	Number of Cyclic Dependencies
NUMA	**:**	Non-Uniform Memory Access
MCL	**:**	Markov Cluster Algorithm
MPI	**:**	Message Passing Interface
OpenMP	**:**	Open Multi-Processing
OS	**:**	Operating System
RBSD	**:**	Ratio of Bridge to Source Dependencies
RCD	**:**	Ratio of Cyclic Dependencies
RDD	**:**	Ratio of Dependency Directions
REIBD	**:**	Ratio of External/Internal Bridge Dependencies
REIID	**:**	Ratio of External/Internal Island Dependencies
RSBC	**:**	Ratio of Sibling Bridge Classes
RSD	**:**	Ratio of Singular Dependencies
SACS	**:**	Same Core Scheduling
TRW	**:**	Abbreviation for TRWeb. A 4 dual-core processor server used in experiments.
UML	**:**	Unified Modeling Language
ZEB	**:**	Abbreviation for Zebella. A 2 hex-core processor server used in experiments.

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\rho_{\mathbf{i}}$	:	Performance of i^{th} experiment.
$\rho_{\mathbf{n}}$	:	Normalized performance.
$\mathbf{D_{out}}$	:	The number of direct dependencies of a class towards other classes.
$\mathbf{D_{in}}$	:	The number of direct dependencies towards a class.
$\mathbf{D_{tot}}$	:	The number of direct dependencies a class have.
$\mathbf{D_{sng}}$	:	The number of dependencies towards a class where the originating class has only one dependency.
$\mathbf{D_{cyc}}$	:	The number of direct dependencies a class have towards itself.
$\mathbf{D_{ext}}$	:	The number of dependencies a group of classes have towards other classes.
$\mathbf{D_{int}}$	:	The number of dependencies towards a group of classes.
$\mathbf{D_{src}}$	:	The number of dependencies between the source classes of a bridge and the classes in the bridge pattern.
$\mathbf{N_{par}}$	:	The number of different ancestor classes that the classes of a bridge pattern have.
$\mathbf{N_{bdg}}$	:	The number of classes inside a bridge pattern.
$\sigma[\ldots]$	:	Standard deviation of a series.
$\omega_{\mathbf{i}}$	:	A core of a processor.
$\mu_{\mathbf{0}}$	:	Main memory.
$\mu_{\mathbf{i}}$	:	A cache memory in a processor.
$\mathbf{b_i}$	:	A bridge pattern instance.
$\mathbf{i_i}$	:	An island pattern instance.
$\mathbf{a_i}$	:	An authority pattern instance.
$\mathbf{h_i}$	:	A hub pattern instance.
$\mathbf{y_i}$	:	A cycle pattern instance.
${}_{\mathbf{p}}\mathbf{c_i}$	:	i^{th} class in a pattern p.
δ	:	Distribution factor of a pattern.
$\mathbf{N_c}$	:	The number of classes inside a pattern.
$\mathbf{N}_{\omega}$	:	The number of cores in a processor.
$\varepsilon_{\mathbf{l}}$	:	Left threshold that distribution can deviate.
$\varepsilon_{\mathbf{r}}$	:	Right threshold that distribution can deviate.
$\uparrow(\mu_{\mathbf{i}})$	:	Function to select one upper level of μ_i in processors' memory hierarchy.
$\downarrow(\mu_{\mathbf{i}})$	:	Function to select one lower level of μ_i in processors' memory hierarchy.
$\Delta()$	:	Function to schedule each object in a pattern distributing them over a set of cores.
$\Pi()$	:	Function to schedule objects in a pattern over a set of cores using operating system's scheduling policy.

MODEL BASED PARALLELIZATION OF OBJECT ORIENTED SOFTWARE FOR MULTICORE SYSTEMS

SUMMARY

As multicore processors are becoming more wide-spread, leveraging of parallelism is once again becoming an important concern during the software development process. Substantial refactoring is required to parallelize legacy sequential software in order to exploit the advantages offered by parallel processing. In this thesis study, guidelines are offered to aid in parallelizing and scheduling of object-oriented programs by analyzing their designs as represented in UML class diagrams.

As a starting point, often occurring patterns of class-dependencies are defined and their characteristics in class diagrams are demonstrated by investigating their properties. Example instances exhibiting the usage of these patterns in class diagrams are presented through analyzing the runtime aspects of these instances. This way, it is possible to identify how they impact the parallelization of object oriented software. Taking these lessons into account when refactoring existing object-oriented software can significantly reduce time and effort required. Proposed methods are evaluated by applying it to three popular design patterns and a real-world case study.

The dependency patterns defined in thesis studies can be detected automatically by using clustering methods and some supporting algorithms. Five different pattern types(authorities, hubs, cycles, bridges and islands) can be detected using class diagram analysis. However the properties of detected pattern occurences can still show a great variance when such a grouping is used. There still exists a need to distinguish each pattern instance regarding different properties they have. Software design metrics can be used to further identify the relation of dependency pattern classes among each other and with the outer group. A metric set is proposed to elaborate the dependency pattern definitions allowing the developer/designer to further identify characteristics of each pattern.

Later in the studies, automatically detecting dependency patterns in software designs is focused. After applying graph clustering techniques to dependency graphs extracted from class diagrams it has been found that these techniques were not able to detect key dependency patterns that relied on characteristic relationships of classes within a cluster to classes outside of that cluster. An algorithm is proposed to detect such dependencies. Experiments show that this algorithm not only detects these elements, but also improves on the studied graph clustering techniques when applied to dependency analysis of class diagrams.

In the last part of the thesis, leveraging utilization of the shared caches of multicore processors is explored. Providing a scheduling mechanism that maximizes throughput by reducing miss-rates of shared caches and preserves the fairness of processor usage is in the center of this problem. Proposed scheduling algorithms in this field usually take advantage of thread level proper- ties of software providing modifications at operating

system level. In the last chapter, a different approach is applied by using software models to guide operating system to effectively map software's objects onto processor cores. The scheduling method takes class dependencies into account and tries to schedule objects of coupled classes onto cores that share the common cache. Firstly, case studies on implementations of three software design patterns(Strategy, Visitor and Observer) is presented. Later, an image filtering software implementation is used in our experiments on two different multiple multicore processor architectures. During the experiments cache-aware scheduler is used in guiding Linux's completely fair scheduler(CFS) and `O(1)` scheduler to perform more cache-aware thread assignments and increase performance. Obtained results promise that guiding/restricting OS scheduler using class-relational information present in the object oriented software model can be fruitful in increasing software performance on multicore processors.

The two main contributions of this thesis are the use of static object oriented software designs in detecting impilict parallelism in software and using this model based information during scheduling of object oriented parallel software. The derivation process of proposed methods are mainly based on re-using patterns that can be found in software designs letting us preserve the software quality during parallelization process. In addition to this process "performance" -as the distinctive quality concern for parallel software- is improved using the derived techniques. The experiments show that performance improvements up to 30% can be achieved using model based techniques.

ÇOK ÇEKİRDEKLİ SİSTEMLER İÇİN NESNEYE DAYALI YAZILIMLARIN MODEL TABANLI PARALELLEŞTİRİLMESİ

ÖZET

Çok çekirdekli işlemciler yaygınlaşmasını sürdürdükçe, yazılım geliştirme sürecinde paralelleştirme çalışmalarının önemi de gitgide artmaktadır. Halihazırda bulunan sıradüzensel çalışma prensiplerine göre hazırlanmış yazılımların paralel işletimin nimetlerinden faydalanabilmesi için önemli bir yeniden düzenleme çalışması yapmak gerekiyor. Bu tez çalışmasında, nesneye dayalı yazılımların paralelleştirme çalışmalarında kullanılmak üzere kullanılabilecek ana hatlar, UML sınıf çizenekleri ile temsil edilen yazılım tasarımları üzerinde yapılan analizler sonucunda elde edilmektedir.

Başlangıç noktası olarak sınıf bağımlılıkları arasında sıkça ortaya çıkan örüntüler ve bu örüntülerin yazılıma özgü gösterdiği karakteristikler, örüntülerin bir takım özellikleri incelenerek ortaya çıkarılmıştır. Bu örüntülerin sınıf çizeneklerinde ortaya çıkma biçimleri, örüntülerden çeşitli örnekler sunularak ve bu örneklerin çalışma zamanında gösterdiği davranışlar incelenerek açıklanmıştır. Bu şekilde bağımlılık örüntülerinin nesneye dayalı yazılımların paralelleştirilmesine olan etkisi incelenmiştir. İncelemelerde ortaya çıkan edinimlerle halihazırda bulunan nesneye dayalı yazılımların paralelleştirilmesi için harcanan çaba büyük oranda azaltılabilir. Önerilen teknikler üç yazılım tasarım kalıbı ve gerçekte de kullanılan bir yazılım üzerinde uygulanarak tekniklerin geçerliliği incelenmiştir.

Tanımlanan bağımlılık örüntüleri öbekleme teknikleri ve bir takım ek tekniklerle otomatik olarak sınıf çizenekleri içinde bulunabilir. Bu tür analizlere dağıtım sınıfı, otorite, döngüsel sınıf, köprü ve adacık ismi verilen beş farklı örüntü algılanabilmektedir. Ancak bu şekilde gruplansa dahi, aynı gruba dahil bağımlılık örüntü örnekleri sahip oldukları özellikler açısından büyük farklılıklara gösterebilmektedir. Her bir örüntü örneğini diğerinden ayıracak bir ölçüm sistemine bu noktada ihtiyaç duyulmaktadır. Yazılım ölçütleri bu amaçla daha detaylı bir analiz sağlamak için kullanılabilirler. Bağımlılık örüntülerinin yazılımda gösterdiği özellikleri detaylandırmak amacıyla bir ölçüt kümesi tez çalışmalarında önerilmiştir.

Tez çalışmalarının sonraki bölümlerinde bağımlılık örüntülerinin otomatik olarak yazılım tasarımlarında algılanılmasına odaklanılmıştır. Bu amaçla yazılım tasarımlarından edinilen çizgeler üzerinde öbekleme algoritmaları uygulanmış ve sonuçta bu algoritmaların özellikle “köprü” adı verilen örüntüleri algılamakta yetersiz kaldığı görülmüştür. Bu sorunu çözmeye yönelik tanımlanan algoritma ile hem “körprü”lerin algılanması sağlanmış hem de böylece öbekleme tekniklerinin bağımlılık kalıbı algılama amacıyla başarımı arttırılmıştır.

Tez çalışmalarında son olarak çok çekirdekli işlemcilerde ortak kullanılan cep belleklerin paylaşılmasından sağlanan faydanın model tabanlı tekniklerle arttırılması üzerine yoğunlaşılmıştır. Bu noktadaki sorun işlemci kullanımında adaleti koruma

amaçlı yapılan iplik-işlemci atamalarının paylaşılan cep belleklerde bulunan verilerin sıklıkla yer değiştirmesine yol açmasıdır. Bu konuda yapılan çalışmalar genellikle işletim sistemi çekirdeği düzeyine yakın değişklik veya ekler içermektedir. Son bölümde bu sorunu çözmek için farklı bir yol tercih edilerek yazılım modeli analizi sonucu ortaya çıkarılan sınıf bağımlılıkları kullanılarak yazılım mimarisi ve işlemci mimarisi eşleştirmesi yapılmıştır. Bu eşleşmeye göre oluşturulan iş dağıtım yöntemi aralarında yüksek bağımlılık bulunan sınıfları, paylaşacakları veri miktarı çok olabileceğinden ortak cep bellek kullanan çekirdeklere atamaya çalışmaktadır. Önerilen iş dağıtım yöntemi “strateji”,“ziyaretçi” ve “gözlemci” isimli tasarım kalıplarının gerçeklemeleri ve bir görüntü filtreleme yazılımı üzerinde denenmiştir. Deneyler sırasında önerilen dağıtıcı Linux işletim sisteminin CFS ve `O(1)` isimli iki farklı iş sıralayıcısını yönlendirmekte kullanılmış ve bu sayede başarımlarını arttırdığı gözlenmiştir. Elde edilen sonuçlar işletim sistemi iş sıralayıcısının yazılım modelindeki sınıflar arası ilişkiler göz önünde bulundurularak cep bellek kullanımını arttıracak şekilde yönlendirmesinin başarımı arttırdığı yönündedir.

Tez çalışmaları sonucunda iki farklı alanda katkılar sağlanmıştır. Bunlardan ilki yazılım tasarımları kullanılarak yazılımın genelinde gizli bulunan paralelliğin ortaya çıkartılması. İkincisi ise model tabanlı bilgiler ışığında nesneye dayalı yazılımların iş sıralamasının yönlendirilmesidir. Önerilen tekniklerin oluşturulması esnasında yazılımlarda bulunan örüntülerin/kalıpların tekrar kullanımı ile paralelleştirme sürecinde yazılım kalitesinin korunması sağlanmıştır. Ayrıca kullanılan tekniklerle paralel yazılımlar için en önemli kalite isterlerinden olan başarımın arttırılması sağlanmıştır. Sunulan deneyler, önerilen model tabanlı tekniklerin kullanımı ile %30’a varan başarım artışının sağlanabileceğini göstermektedir.

1. INTRODUCTION

Originally the term computing is used for counting and calculating. People who perform those operations are called computers. But with the spreading of computing machines after the pioneering work of transistors by W. Shockley et al. in 1947, the discipline of computing has begun to be defined as "*the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. Actually, the fundamental question underlying all computing is 'What can be (efficiently) automated?'*"([1] pg.12). In terms of effectiveness, one of the first aspects that comes into mind is collaboration of many processing elements working on a problem divided into sub problems; in other words parallelization. Parallel computing can be seen as an evolution of serial computing that attempts to emulate many complicated, interrelated events happening at the same time, yet within a sequence. First practical usage of parallel computing in computer industry was ILLIAC IV in 1976 which used up to 256 processors to provide an efficient level of computation. Until now parallel computing is being realized in many different platforms ranging from cluster computing to distributed computing. In the architecture level, Flynn's taxonomy [2] classified processing levels based upon the number of concurrent instruction (or control) and data streams available.

Until the development of multi-core CPUs, parallel computing is widely used in servers which use multiple processing units in separate chips interconnected via external buses. The term multiprocessing is used for the ability of a system to support more than one processor and/or the ability to allocate tasks between them [3]. However as the processor technology comes closer to the edge of the Moore's law microchip producers has begun to search for alternative ways to improve computing efficiency. As a result they recently come up with the concept of multi-core computing. A multicore CPU (or chip-level multiprocessor, CMP) combines two or more independent cores into a single package containing a single piece silicon integrated circuit (IC), called die, or more dies packaged together. A multicore microprocessor implements multiprocessing

in a single physical package. A system with N cores is effective when it is presented with N or more threads concurrently. In this context, “multi” typically means a relatively small number of cores. However, the technology is widely used in other areas, especially those of embedded processors such as network processors and digital signal processors, and in GPUs. Despite many advantages of parallel computing, multicore CPUs face a very difficult disadvantage in terms of legacy applications and software needs. Those disadvantages include the requirement of operating system (OS) support and adjustments to existing software to maximize utilization of the computing resources provided by multicore processors. Also, the ability of multicore processors to increase application performance depends on the use of multiple threads within applications.

The studies in this thesis are directed towards increasing the performance of object oriented software that runs on multicore processors. As the processor technology continues increasing the number of cores in a single die, problem of decomposing software for parallel run becomes graver as well. Until the last decade, most of the parallelization efforts for solving this problem were aimed towards expensive multiprocessor hardware and its programmer. As a result, there exist a wide area of study that focuses on low level programming models that can be applied at programming stage of software development by a limited number of elite programmers. The studies in this thesis is aimed towards modern driven parallelization of object oriented software in order to present a basis for parallel decomposition that can be applied by a wider range of practitioners at early stages of software development.

1.1 Multi-core Computing

Chip multiprocessors, also known as multicore computing, involves more than one processor placed on a single chip and can be thought of the most extreme form of tightly-coupled multiprocessing. In fact Intel and AMD have recently announced their multicore processors for PC usage. This is the first time when parallel programming methodologies are brought down from High Performance Computing to PC user and developer level. Actually a multicore CPU is similar with multi processor supercomputer processor architecture, except the processors are produced in a single

die physically. In Dual-Core architecture (like Intel's) as seen in Figure 1.1 there exist two processing elements with their own caches and a shared cache. These components are connected to each other via internal buses. In AMD Opteron dual-core architecture the main principles of NUMA is used in memory management.

It can be seen that multicore processors don't bring any new concepts to parallel computing models. It is more or less possible to use existing parallel programming models like OpenMP and MPI with multicore systems. But there exists two issues of this situation. The first issue is the efficiency of the previous programming models for application software development. It is clear that much of the effort for parallel computing is based on high performance computing and problems which are a little bit more data centric. Secondly, with the introduction of heterogeneity, current parallel programming models has begun losing efficiency in adapting complex core structures.

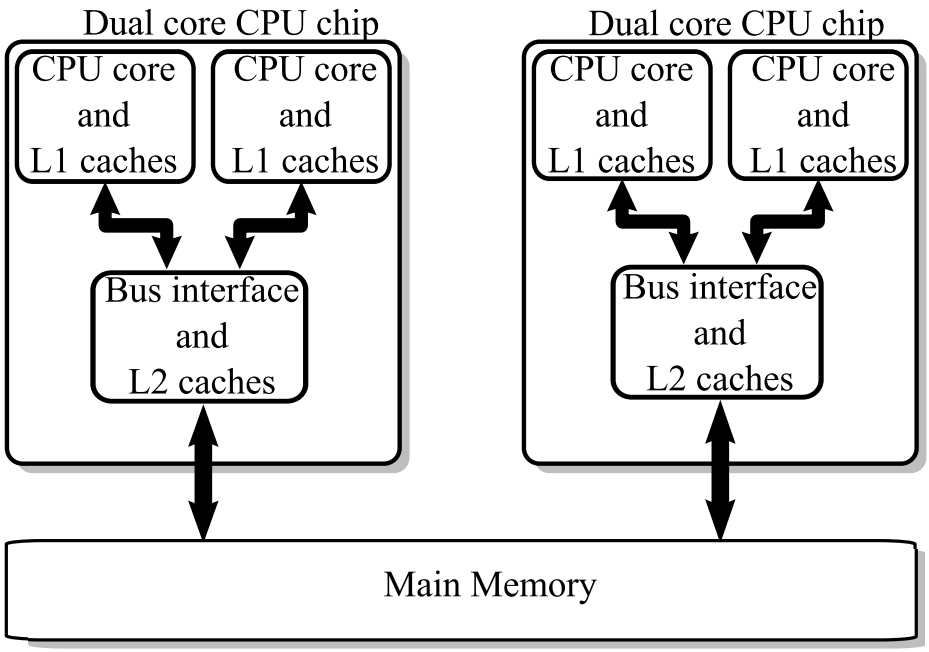


Figure 1.1: Diagram of a typical multiple multicore processor system.

Since the emerging of the multicore CPUs, programming environment didn't change much because dual cores perform well in multitasking and thus they barely meet the expectations. But in the future (and even in today by quad-core users) matching Amdahl's law, multitasking performance will not be high enough. This is because the user will not be able to produce as much tasks as the number of CPUs at the same time. This will result in computer performances equivalent to the speed of a few cores

but far less than the sum of all available. At this stage called manycore computing we need to perform the parallel decomposition while programming the computer rather than leaving it to the operating system.

The future of multicore computing is called manycore computing, where hundred or even thousands of cores are interconnected in a variety of different ways, having different number and sizes of cache memories. This can produce an exploitation in the design space of processors and bring many different problems that software developers haven't expected to deal with before. One of the most serious problems is producing scalable software that is going to meet performance expectations in a variety of different processor architectures. Viability of producing such an effective solution to this problem is beyond the scope of this discussion, on the other hand one may expect to deal with a heavy load of refactoring effort in such a heterogeneous environment.

Refactoring code for every different processor architecture is an unbearable burden. However, the perspective of today's software engineering discipline is converging towards automatic code generation and architecture software factories, utilizing model driven engineering methods. It is possible to expect that, parallelization for different processor architectures is going to take place among these efforts as the multicore processors continue to develop.

1.2 Model Based Parallelization and Scheduling

In order to take advantage of multiple processors, sequential legacy software needs to be refactored: it can be parallelized by injecting threads into the code and localizing them on different processors or by distributing objects across processors. Doing so is cumbersome since it requires intimate knowledge of the legacy software and detailed code analysis.

Exploring parallelism implicit in software requires global analysis of the system which is difficult by focusing on code alone. Consider the example of a web browser originally designed for a single processor system: Say, this browser supports tabbed browsing and uses a single address bar and navigation toolbar for all of its tabs. A key decision when redesigning the browser for a parallel environment might be to implement the tabs as separate threads while keeping address bar and navigation

toolbar shared sections among the tab threads. Alternatively, the address bar and navigation toolbar could be embedded within the tabs and handled as part of a tab thread.

How to realize such opportunities of parallelization by examining the source code and looking for parallelizable constructs is not obvious. For object oriented software in particular it is not always possible to find large bodies of consecutive instructions within a single class, as methods are often purposefully kept short. Moreover, just considering the interrelations between a few classes will not be helpful in this example since all of these components–tool bar, address bar, and tabs–may be involved in complicated class relations and class hierarchies provided by the leveraged framework. Much unnecessary detail will be present in the code making analysis by code inspection and refactoring difficult.

It may be more efficient to explore parallelization opportunities at a global scale by relying on software models rather than by relying on code inspection. While due to its familiarity it may be obvious to consider tabs as an opportunity to introduce parallelism, there may be other less obvious opportunities implicit in the design. For less familiar domains, finding such opportunities will be more difficult. In addition, model analysis is subjective and different designers inspecting the model may arrive at different interpretations. Proposed approach attempts to overcome these difficulties by identifying recurring structures in software models that have a bearing on parallelization and then provide recipes for how to leverage these structures for parallelization.

During thesis studies, UML class diagrams are chosen as the starting point to explore the parallelism implicit in the structure of a software system. This may be unintuitive, as class diagrams model the static structure of software. However class diagrams also give important information about the runtime behavior of the modeled system. By analyzing concepts such as relationship cardinalities, dependency sequences, and inheritance relations, one can draw inferences about possible manifestations of runtime patterns these classes participate in.

Another key point in thesis studies is using software design patterns to propose reusable and scalable approaches for parallelization. Most of the time, proposed techniques are

experimented on intuitive implementations of software design patterns. The reason behind this decision is the place of design patterns in today's object oriented software. Firstly, design patterns tend to be used frequently as the building blocks of software design being more specific and descriptive in software design than classes but also being modular enough to be used in a versatile way. Secondly, design patterns are seen crucial in producing quality software because they are continuously improved through time.

On the other hand using static UML class diagrams in making decisions that are going to effect scheduling is not a heavily studied topic. Especially static models are known as quite disconnected with the runtime behavior of software. However class diagrams reflect the solution domain of the problem carrying information like possible data sharing components of the software. Such information can be useful in parallelization and scheduling of software because placing the parallel components of software to allow effective data communication between them may be as important as an effective parallelization. Based on these reasons using a model-driven pattern based methodology has helped to preserve the overall quality of the system whilst improving performance of the software.

As a summary, processor technology has undergone a serious change in the last decade by the introduction of multicore processors. Sooner or later, it is expected to experience such an evolution in software technology as well. In this thesis, model based approaches are proposed to be used through the evolution of object oriented software. The possibility of preserving the overall quality while improving software performance is discussed and the experimental results related to such discussions are presented as main contributions.

1.3 Contributions and Dissertation Outline

Considering the expected shift towards integrating parallelization inside sequential developed software, thesis studies are focused on developing methodology for refactoring of sequential software for parallel systems using software models as a medium. Instead of developing new programming models for parallel software development, main focus is aiding software designers and developers in exploring implicit parallelism that reside in software and steering operating system in a proper

way to enhance the data reuse between software components. Two main contributions are presented in this dissertation towards this goal are:

- Model driven parallelization: The difficulties in using classes and/or conventional design patterns are discussed. To overcome these difficulties a set of structural patterns are proposed which emerge in class diagrams caused by the graph based nature of the diagram. Following the definition of dependency patterns, metrics are also proposed to identify their properties and also clustering methods are discussed in detection of dependency patterns.
- Model driven scheduling: Shared data and cache utilization of processors are not yet handled by the modern operating system schedulers. Since dependency patterns also capture possible common data usage among software components an enhancement to apply a more cache aware scheduling is proposed using dependency patterns and their properties.

The rest of the thesis is organized as follows: Chapter 2 contains previous studies as a basis to thesis studies on model driven parallelization and scheduling. In Chapter 3, dependency patterns are presented as recurring structures in UML class diagrams and their role in software parallelization is discussed. A metric set is also presented Chapter 3 to perform finer-grained analysis on dependency diagrams. Finally the usage of current clustering techniques in detecting dependency diagrams are discussed and an improvement to clustering techniques is proposed. An enhancement to scheduling of object oriented software is proposed in Chapter 4. As the initial phase, experiments of a cache-aware scheduler on design patterns are presented. Later an object dispatcher implementation based on cache-aware scheduling methodology is proposed and performance improvement gained by its application on an image filtering software is discussed. The thesis is concluded by summarizing the achievement and giving future directions in improving quality of object oriented software for multicore processors.

2. RELATED WORK

In this thesis model driven methods have been used on improving quality of object oriented software for parallel systems. The studies performed in this purpose have been structured as two main parts which are based on model driven parallelism exploration(Chapter 3) and model driven scheduling improvement(Chapter 4). Many different studies exist in the literature that forms a basis for our work and stand as complementary approaches.

2.1 Model Driven Parallelization

Parallelization of Object Oriented Software is applied at many different stages of software development. At program code level, [4] focus on automatically exploiting implicit parallelism in loops and multi-way recursive methods. They have restructured a Java compiler to specify implicitly parallel structures like loops in an explicit way. Another transformation based study that parallelize loops to improve performance is by [5] where the transformations can be applied by an OpenMP compiler for Java like JOMP [6] to exploit code level parallelism. [7] worked at the bytecode level to provide mechanisms to parallelize Java applications and execute them on distributed processors, without requiring the application programmer to explicitly use dedicated message-passing libraries. In [8] and [9] an automatic parallelizing system based on Java is designed and implemented where dependencies in the source code are analyzed for implicit functional parallelism. On other approaches [10, 11], in addition to code analysis, other environments such as compilers, run-time environments, operating system kernels, etc., are utilized to exploit the implicit parallelism in object oriented software.

At the model level, behavioral models (such as UML behavioral diagrams) have been used to reason about different aspects of parallelizing object oriented software. Sequence diagrams have been leveraged in timing analysis, synchronization and deadlock detection in concurrent and distributed object systems [12–14]. Activity

diagrams have been used to analyze timing properties [13, 15]. [16] transformed UML statechart diagrams to PROMELA specifications in order to apply the SPIN model checker [17, 18]. Analysis need not be restricted to a single type of behavioral diagram; different UML diagrams may be included in the analysis [19–21]. Parallelizing software based on its sequential model has also been studied for embedded systems. For instance, [22] used a UML-based code-block-level modeling language to perform containment-checking-based methodology for application partitioning verification for multiprocessor embedded systems.

Instead of using code analysis and dynamic models/object models in parallelizing object oriented software, we analyze static class diagrams. These diagrams can be obtained at the earliest stages of software design. [13] took advantage of stereotypes applied to elements of class diagrams to aid in detecting deadlocks in distributed object systems. [23] embedded an explicit CSP notation in UML class diagrams. [24] used a graph model of the relationships between events created by the execution of a distributed system to derive a model of the concurrent relationships in the same system. A similar graph-based approach is used in this thesis studies to reason about parallelism but instead of event relations class relations are used. [25] utilizes use cases to check whether the behavior of an entity complies with the composed behavior of its sub-entities.

[26] proposed a system called COMPASS, providing guidelines on parallelization process based on the former techniques applied by developers during parallelization. [27] proposed a reverse engineering based method to facilitate systematic migration of code from sequential to parallel processing environments. Their approach constructs dependency graphs of FORTRAN programs and uses rule based methods in parallelization.

Besides dependencies amongst software components, utilization of caches is another important topic when parallelizing software for multicore/multiprocessor systems. Cache locality is a well-studied problem which also gains importance as multicore processors are becoming prevalent. An important area of research is modifying scheduling mechanism of operating systems in order to take advantage of cache memory, see [28–33].

In this thesis, parallelization solely based on examining interrelations among classes is focused in Chapter 3. The guidelines on dependency pattern parallelization point to the possible regions in a *software model* that should be examined when parallelizing the software. Dependency graphs obtained from static class diagrams of object oriented software are used to reveal recurring structures in software models and to reason about their parallelization. Following the presented approach, it is possible to identify areas in the model that might benefit from parallelization. Presented guidelines aid to structure the code derived from the areas of the model that have been pinpointed by class diagram analysis, aiming to obtain performance improvements from parallelization. Class diagrams can be used at the design stage before the system is implemented or after reverse engineering of the code of a sequential legacy software has been performed.

2.2 Model Driven Scheduling

For the last decade, mainstream in processor technology is chip multiprocessors, also named as multicore processors, which involve multiple processing cores in a processor die. By their nature, multicore processors utilize parallel running software where cores are assigned to each thread produced by parallel decomposition of software. This assignment operation is done by operating system schedulers, which put emphasis on fairness and load-balancing problems rather than utilization of shared data among threads.

As multicore architectures get more complicated, cache memories not only serve as buffers for accelerating memory access of threads but also provide a rapid communication medium for shared data among threads. Recent multicore processor architectures contain relatively smaller caches for each distinct core and larger shared caches for the cores that reside on the same chip. It can be expected to encounter more complicated cache hierarchies as the number of cores increase.

Aside from this situation, current operating system schedulers do not provide an effective way to deal with cache utilization of processors yet. Instead, their primary concern is more fair time-slicing of processing elements to provide user balanced running time of applications [34–37]. This is quite natural since operating system scheduler is expected to run on a wide range of processor architectures and application

software. Leveraging different concerns in such a heterogeneous environment is a serious challenge, that becomes more important as multicore processors continue evolving towards manycore processors.

Improving operating system schedulers to take cache utilization into account is being heavily studied by the community. In most of the studies, a single centralized solution to replace the scheduler is proposed using data gathered from runtime profile of software [28–33] [38–40]. Since proposed improvements are at operating system level, software analysis are carried on lower level software structures like loops or thread groups.

Using graph based techniques on scheduling has been applied in a variety of different cases. Earlier studies on using graph matching algorithms for parallel scheduling was applied on multi-processor architectures. In [41], task graphs were used to identify special tasks they call “backbone” tasks that carry the application. Using those special structures they tried to map the task graph effectively onto a multiprocessor system. Discovering special structures inside graph models of software forms the roots of the studies in this thesis. Trifunovic and Knottenbelt used graph coloring to effectively decompose parallel sparse matrix–vector multiplication algorithm [42]. Further information on utilization of task graphs in scheduling can be found in [43].

Later when the chip multiprocessor began to emerge, studies on scheduling by graph matching began to focus on locality aware scheduling. In their paper, Guangyu Chen et al. performed data aware scheduling in four steps [44]. In the first step, the application code is parallelized and the resulting parallel threads are assigned to virtual processors. The second step implements a virtual processor-to-physical processor mapping. In the third step, data elements are mapped to memories attached to CMP nodes. The last step of their approach determines the paths (between memories and processors) for data to travel in an energy efficient manner. This strategy partly resembles the software decomposition and mapping strategy presented in Chapter 3 except the process of the amount of data being shared at the third and fourth steps of their study. A more cache specific study is performed by [45]. They propose a new cache management policy called Promotion/Insertion Pseudo-Partitioning (PIPP). Instead of explicitly partitioning the cache by ways, sets or total occupancy, PIPP implicitly partitions the cache by simply managing the insertion and promotion policies of the cache.

One of the latest studies that model the resources of the processor and the software to perform matching between two models for multicore systems is done by [46]. In their paper they introduce the Multi-BSP model to model all levels of an architecture together. Later at each level, their Multi-BSP model incorporates memory size as a further parameter. The goal of the study is to identify a bridging model on which the community can agree, one which would influence the design of both software and hardware. The proposed architectural model in Section 4.2.1 is a subset of his general model which doesn't take communication costs.

Previous work on cache-aware scheduling on multicore systems generally takes advantage of dynamic information of software provided by runtime analysis [28–33]. This type of scheduling can be supported with the information obtained by static analysis of software models and shared data between them. Wickzier et al. provide annotations for the programmer to explicitly guide their O^2 scheduler called CoreTime in managing shared data among multiple threads [38]. Xue et al. also proposed a method claiming that static scheduling can be made locality aware by ensuring that the set of iterations assigned to a processor exhibit data reuse [39]. In Chapter 4 a further step has been taken and the impact of inter-class relationships of software's object oriented model is evaluated to guide its scheduling.

Another interesting point in Xue's study is the usage of loops as recurring software components in scheduling decisions. Loops are heavily used in software parallelization/cache-utilization studies before. Tam et al. utilize threads as disjoint components of parallel/concurrent software and schedule them based on sharing patterns they pose at runtime [30]. In other words, they basically find coupled threads at runtime and schedule them to share L2 caches. Federova et al. identify coupled threads as co-runner threads and try to reduce performance variability caused by cache-unfair scheduling of them [40]. In thesis studies coupled software components at object oriented level has been focused and the data sharing classes' objects (which are already specified at software model/code) are used to guide the operating system's scheduler.

Using static software models is another rarely used subject in cache-aware scheduling studies. One of those studies that explicitly uses models and software abstractions in maximizing cache reuse in multicore scheduling is done by [47]. They try to solve

optimal multicore scheduling problem by using a graph theoretic formulation and answer set programming in their study. In this thesis object oriented software models are specifically used to reason about data sharing among software's classes.

3. EXPLORING IMPLICIT PARALLELISM IN CLASS DIAGRAMS

In this chapter, analysis on static class diagrams of object oriented software is going to be focused and the impacts of the recurring structures detected by those analysis on the parallelization process is going to be presented. Main goal in identifying those recurring structures is gaining insight on characteristics of the software by relying on the software model at hand. This way it can be possible to analyze software characteristics from a parallelization perspective and use the detected recurring structures in parallelizing object oriented software.

In Section 3.1 the relation between class diagrams and the parallelization process of object oriented software using class diagrams will be examined briefly. In Section 3.2 dependency patterns will be presented as the frequently recurring structures in class diagrams and their impacts on the paralellization performance will be presented. Examining the general properties of dependency diagrams further, a metric set will be presented in Section 3.3 to allow better distinction among dependency patterns. Finally in Section 3.4, two methods will be presented to detect dependency patterns inside class diagrams.

3.1 Class Diagrams and Implicit Parallelism

Refactoring legacy application software is a crucial step in introducing the concepts of parallelism into today's software development efforts. When analyzing object oriented software for parallelism, using diagrams that model classes and objects as well as the relationship between them can be fruitful [48, 49]. Behavioral diagrams such as sequence diagrams [13, 50] and activity diagrams [51] were used to reason about timing aspects of object oriented software.

UML provides different specification techniques and diagrams to model the various aspects of a software system. For example, static class diagrams model the classes used in the software and the many kinds of relations that may exist between them;

sequence and communication diagrams represent the dynamic structure of the software by specifying message exchanges between objects. Therefore it would be quite natural to investigate the dynamic structure of the software using a behavioral model in order to detect opportunities for parallelization.

In practice, behavioral model analysis is not without difficulties. In particular, when looking for opportunities for parallelization in a system-wide scope and attempt to discover these opportunities in a top-down manner, sequence and communication diagrams can be prohibitively complicated. Providing a detailed system-wide scenario or processing a communication diagram consisting of all the classes in software system and the communications between these classes is usually not feasible without applying abstractions. But finding appropriate abstractions or identifying suitable decompositions of the software system is fraught with difficulties also, such as ensuring consistency among different parts of the system model when recombining them at the system level [50, 52, 53]. Without this global system view, it is only possible to perform local analysis using such behavioral models.

Class diagrams have been used together with behavioral diagrams to connect software behavior with software structure. A new relationship has been added between classes to represent behavioral evolution, referred to as “context relation” [54]. Context relations are used to model dynamically related classes at runtime. Two classes are context related if one of them can dynamically affect the behavior of the other. A reflective architecture which provides the ability to change object behavior at run-time by using design-time information was also proposed in the literature [55]. They integrated reflection with design patterns to get a flexible and easily adaptable architecture that can dynamically adapt the software system to environmental changes. In their approach, the system is divided into its structure described by its object model and its behavior described by state and sequence diagrams. Structural evolution is carried out by causal connection between these two layers.

Analyzing static class diagrams to reveal parallelism turns out to be helpful in a variety of ways. Unlike for behavioral diagrams, obtaining a global class diagram of the overall system is possible even without simplifying abstractions, albeit it may be tedious. Further, as a practical consideration, it is much easier to obtain class diagrams from a legacy system using reverse engineering techniques. Obtaining behavioral

diagrams may not even be feasible without a large set of test cases being available such that these test cases cover every aspect of the system behavior. Moreover, reverse engineering a behavioral diagram requires the system to be executed, which may induce a large number of different diagrams based on the scenarios used to execute the program.

In class diagrams, all possible object interactions that can occur at runtime are represented by class relations; for instance an association between two classes means that at runtime an interaction may occur between instances of these two classes. Complete class diagrams represent the system as a whole, while each behavioral diagram represents only a single runtime trace (or several traces, when inline constructs are used). If two classes are unrelated in a class diagram one may deduce that they will not be related at runtime; however, it is not possible to conclude this from communication diagrams. If it is possible to separate two independent regions inside a class diagram, it may be possible to separate those regions at runtime as well. By identifying these regions, parallelizable parts of the software can be discovered.

An important advantage of leveraging dependency patterns in parallelization is that they can be identified automatically using class diagrams. In the refactoring and parallelization process, only identified portions of the resultant code need to be focused on and a global and thorough analysis of the code can be avoided.

In this chapter, dependency patterns found in class diagrams are introduced and their impact and guide on parallelization is illustrated. The impact of dependency patterns on the parallelization of object-oriented application software is studied using several familiar design patterns [56] and a case study of an open source compiler project Jikes [57].

Jikes is a mid-size project consisting of roughly 250 classes and about 30 header files written in C++. In the following experiments, three packages called `CLASS` (39 classes), `LOOKUP` (41 classes), and `AST` (103 classes) were used. Class diagrams for these packages are obtained by reverse engineering from header files, resulting in medium to large size diagrams. Parallelism is injected into the code sections resulting from the design segments identified as dependency patterns.

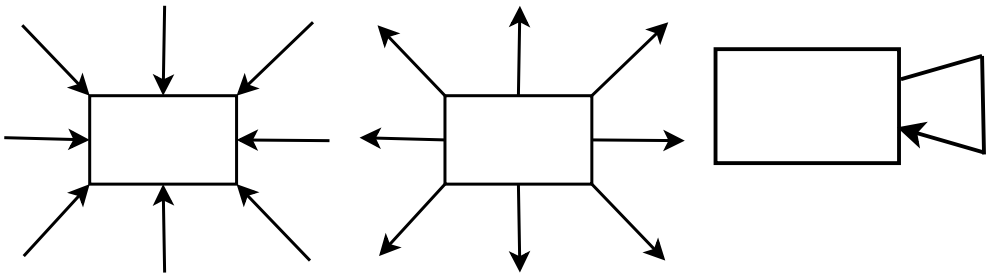
3.2 Dependency Patterns in Class Diagrams

Dependency patterns can be identified using dependency relations extracted from class diagrams. Similar concepts have been introduced by [58] who applied graph theoretic techniques to UML class diagrams. Dependency is defined in the context of this study as any direct usage relation among classes. These relations include associations, as well as access to attributes and method parameters. It does not include composition, generalization, and realization relations; these relations will be considered in later sections.

Dependency patterns may involve a single class and its dependencies to or from other classes, as well as multiple classes and their dependencies between each other and to or from other classes. In the single-class case, a pattern consists of the single class and dependencies to or from other classes outside the pattern. In the multi-class case, dependency relations exist within the pattern in addition to dependencies to or from classes outside the pattern.

3.2.1 Single-class dependency patterns

Single-class dependency patterns fall into the three categories “authority”, “hub”, and “cycle”, based on the type of dependency relationships the class is involved in.



(a) Authority. **(b)** Hub. **(c)** Cycle.

Figure 3.1: Single-class dependency patterns.

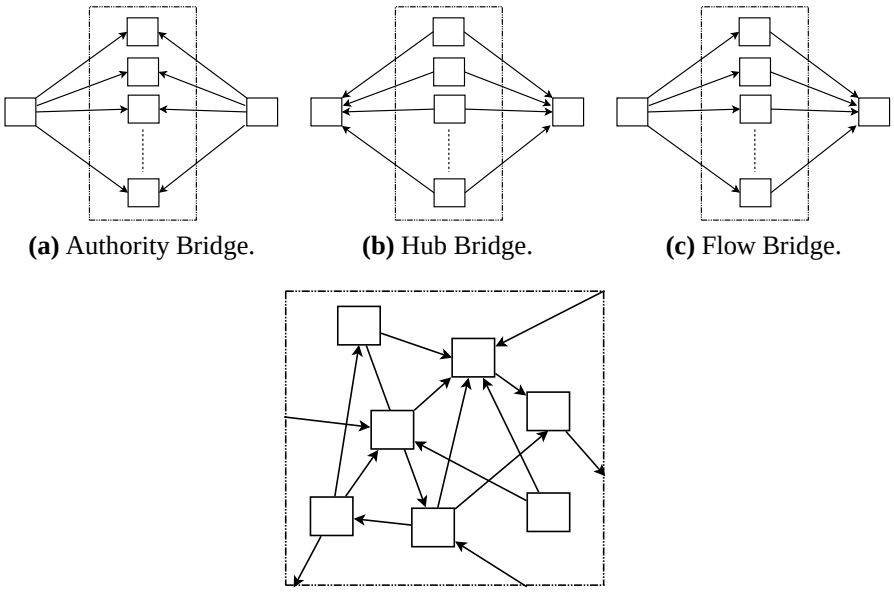
- An **authority** is a class that is involved in a large number of dependencies from other classes to this class, see Figure 3.1(a). In other words, an authority is a class which other classes are coupled to. How many incoming dependencies are required to constitute an authority class is subjective and may differ based on the situation. In order for a class to be identified as an authority, it should have a significant portion of dependencies among all the dependencies present on the class diagram.

- A **hub** is a class that has a large number of dependencies to other classes, see Figure 3.1(b), that is, it is coupled to a number of other classes in a noticeable way. Similarly to an authority class, its identification is subjective and relative to other dependencies found on the class diagram.
- A **cycle** is a class that has a dependency to itself, see Figure 3.1(c). Identifying a class as a cycle is simple as it merely requires detecting a self-dependency.

Authorities and hubs are important in terms of parallelization since they have the potential to be accessed frequently at runtime. Cycles are also important because they show the potential for sequential behavior to be imposed at runtime which needs to be avoided in order to effectively parallelize such patterns.

3.2.2 Multi-class dependency patterns

Multi-class dependency patterns fall into the categories “bridge” and “island”. These categories are formed with respect to the dependency relation that exists within the pattern.



(a) Authority Bridge. **(b)** Hub Bridge. **(c)** Flow Bridge.

(d) Island.

Figure 3.2: Multi-class dependency patterns.

- A **bridge** consists of a group of classes where each class in the pattern has common dependencies to at least two classes. Classes may be members of multiple bridges, and therefore have additional dependencies to other classes outside the pattern. In addition, classes in a bridge may also have dependencies between each other.

(These kind of dependencies are rare in practice.) Bridges come in the form of "hub bridge","authority bridge", and "flow bridge" which result from their relationship to classes outside the pattern. Figure 3.2(a) shows an "authority bridge" where common source classes have dependencies to the classes inside the bridge pattern. Figure 3.2(b) shows a "hub bridge" where all the classes inside the bridge have dependencies to a common set of target classes. In a "flow bridge", as shown in Figure 3.2(c), the classes inside the bridge pattern have dependencies from a common set of source classes and to a common set of destination classes.

- In an **island** pattern, members of the pattern have most of their dependencies within the pattern, see Figure 3.2(d). Islands form clusters in the dependency graph and can be detected using clustering techniques [59–61].

Classes inside an island are strongly coupled with each other and objects of these classes can be assigned to the same (or nearby) processing elements to benefit from cache reuse since they tend to communicate frequently amongst each other. During the experiments it is observed that bridges represent alternating behavior at runtime which often results from polymorphism.

Figures A.1, A.2, and A.3 show the class diagrams for three different modules of the Jikes compiler [57], `CLASS`, `LOOKUP`, and `AST`. In these diagrams, many occurrences of above dependency patterns can be identified easily.

3.2.3 Occurances of dependency patterns

Interpreting dependency patterns based only on their dependency relations does not provide us with enough detail to infer the properties they impose at runtime. Relations like inheritance and composition can provide additional information. For instance, a class may be determined to be an authority since it has many dependency relations to other classes, however this structure does not always imply that the dependencies concern common aspects of the system behavior. On the other hand, knowing that the authority is also at the top of a class hierarchy provides additional valuable information about the pattern: the authority class is being used in a polymorphic way and all dependencies are focused on a smaller set of behaviors. Likewise, detecting that a class as a hub is not enough to immediately allows us to conclude that it can be parallelized since having many dependencies towards other classes does not require

them to be independent. Knowing that a hub creates many objects allows us to infer that the resultant objects can be handled independently from each other. In order to augment dependency relations with other important relationships between classes, [58] considers these relationships as one single relation. However, this analysis results in a loss of precision in terms of inter-class relations.

Examples of such differences in patterns can be seen in the Jikes case study, where the same kind of pattern occurs for entirely different properties. One example involves the classes `AstExpression` and `StoragePool` (see Figure A.3). Although these two classes are determined to be authority classes through dependency graph analysis, they perform very different roles. `AstExpression` is a superclass, representing expressions in the abstract syntax tree (AST) while `StoragePool` is a container class holding different types of AST elements during the compilation process. It is expected that `AstExpression` will be used by a smaller number of class instances than `StoragePool`.

Another example involves hub classes. `ClassFile`, `FieldInfo`, and `MethodInfo` from the `CLASS` module of Jikes (see Figure A.1), are hubs that are at the sources of a bridge and connect mostly with bridge classes. In contrast, the hub class `Control` from the `LOOKUP` module (see Figure A.2), has various kinds of external dependencies to authorities, bridge classes, and classes inside islands. It also constructs and initializes a bigger number of objects than the earlier mentioned hubs.

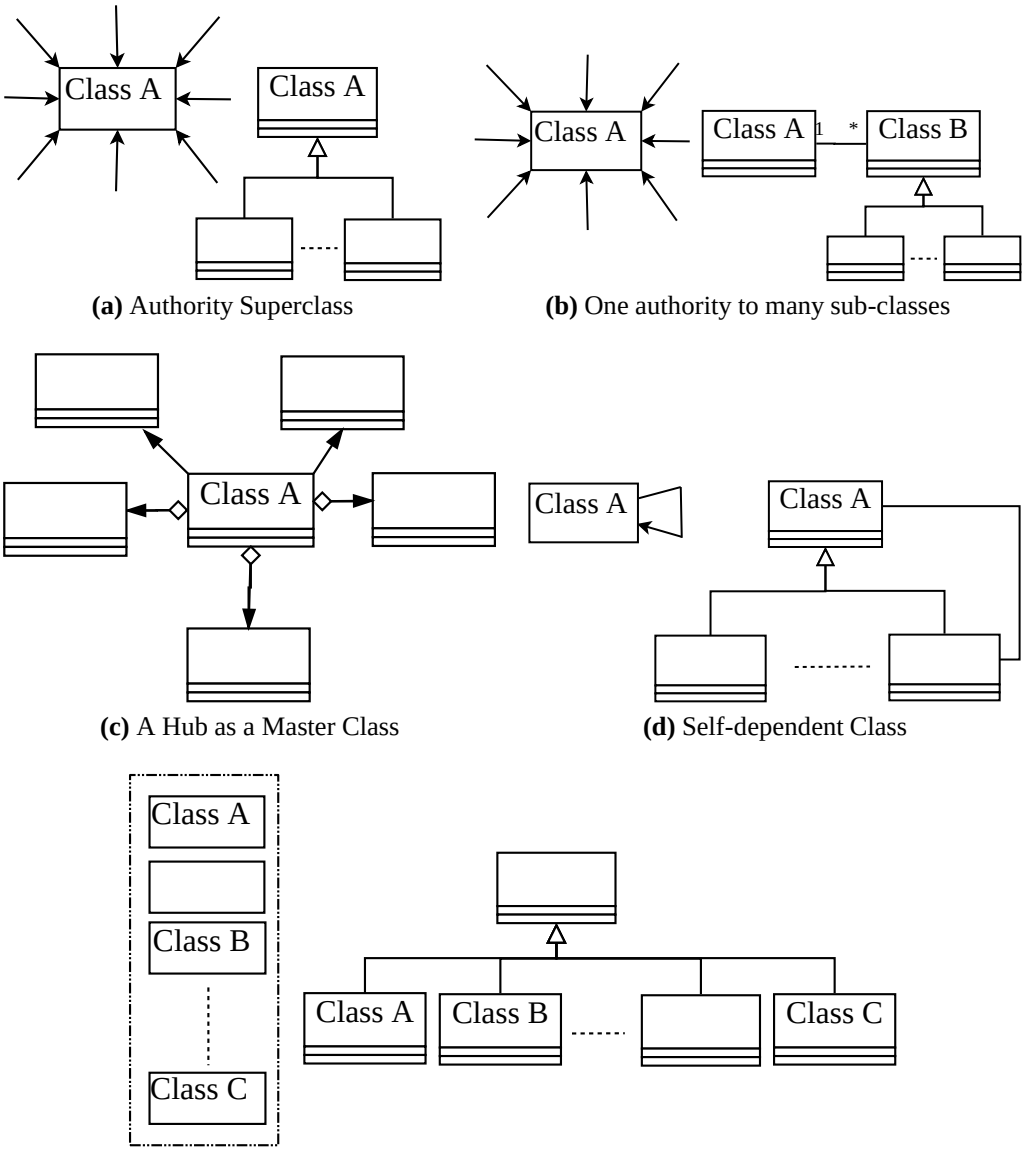
A final example involves two different bridges, again from Jikes. Comparing the bridge shown in the center of Figure A.1 to the bridge shown at the upper left of Figure A.3 consisting of classes `AstForeachStatement` and `AstCatchClause`. The former bridge holds classes that are used in an alternating way during attribute processing when compiling source code. A similar frequent use cannot be found for the latter.

Thus, based on an analysis of dependencies only a number of patterns may be arrived having different roles and properties that affect parallelization differently. Such patterns should be separated by considering additional relations, other than dependency relations. This poses an additional challenge as one may obtain numerous dependency patterns from dependency graph analysis. Examining each occurrence of a pattern one by one and trying to find common properties they represent is tedious and complicated.

Relevant properties of patterns need to be expressed explicitly and consistently to improve the definition of simple dependency patterns.

By studying the typical relationships that classes in dependency patterns partake in, it is determined that hub and authority classes tend to be ancestor classes, tightly coupled to other classes, and mostly use other classes in the pattern. Bridges are typically sibling classes with little or no relation to another class and to classes outside the pattern. Thus the following typical occurrences of dependency patterns in class diagrams are identified. These examples instantiate dependency patterns and also consider relations such as inheritance, association, and composition. The impact of parallelization of these patterns are highlighted for each case.

- An **authority superclass**, see Figure 3.3(a), is placed at the top of an inheritance hierarchy. The following points should be taken into consideration when parallelizing such classes:
  - Authority superclasses encapsulate common information for its descendants. Usually, descendants are used in a polymorphic way.
  - Heavily used portions in an authority superclass that are inherited to its subclasses should be protected against parallel access.
  - Sections of its subclasses that hold common synchronization properties can be abstracted in the authority superclass.
- **One authority to many sub-classes** consists of two classes with a one-to-many relation between them, where the authority class has cardinality 1 and the “many” side of the relationship is a superclass in a class hierarchy, see Figure 3.3(b). The authority must not have any descendants in this case. The following points should be taken into consideration when parallelizing such classes:
  - The authority class becomes a local critical section.
  - There exist independent links towards the authority class form the many side(Class B) which can be executed in parallel.
  - The consistency of the concurrently accessed authority class attributes must be assured.



(a) Authority Superclass **(b)** One authority to many sub-classes

(c) A Hub as a Master Class **(d)** Self-dependent Class

(e) Sibling Bridge Classes

Figure 3.3: Dependency pattern occurrences in class diagrams.

- **A hub class** can be identified as a **master class** if it has many composition or aggregation relations to other classes, see Figure 3.3(c). The following points should be taken into consideration when parallelizing such classes:
  - A hub class uses many objects frequently in order to orchestrate system behavior.
  - Introduce parallelism to the class directly by analyzing independent portions of its methods.
  - Traditional parallelization opportunities like loop parallelization can be spotted easier in this type of class.

- **Self-dependent classes** have dependencies to themselves or to their ancestor class, see Figure 3.3(d). The following points should be taken into consideration when parallelizing such classes:
  - Self-dependencies negatively affect parallelization as they usually impose sequential behavior.
  - Such dependencies should be eliminated by transforming these patterns to parallelizable structures (e.g., by transforming access to a linked list into a table access)
  - Often, these classes tend to include global variables or class variables. Such variables should be eliminated as much as possible.
- For **bridge classes**, some or all classes in a bridge element are siblings in the class hierarchy, see Figure 3.3(e). Following points should be taken into consideration when parallelizing such classes:
  - Bridge classes are frequently accessed in an alternating way, making it possible to introduce parallelism on bridge access. Sections of code that use bridge classes should be parallelized.
  - If the bridge is a hub bridge, instances of the sibling classes can be distributed freely over available processors. The opposite ends of the bridge should be synchronized since they are accessed in parallel by the bridge classes.
  - If the bridge is an authority/flow bridge, instances of the sibling classes should be distributed once and localized (that is, they should not be migrated among processing elements). The rationale behind this policy is to avoid having the processing elements wait for each other in the case where objects of the same class are synchronized during object access.
  - An object distribution policy can be implemented in the ancestor class and can be inherited in descendant bridge classes.

3.2.4 Parallelization using dependency patterns

In this Section, implementations of the Observer, Decorator, and Abstract Factory design patterns [56] are used to demonstrate the parallelization of dependency patterns. Successively, IBM Java Jikes compiler [57] is analyzed as a case study to identify occurrences of dependency patterns in real-life systems and to study their effects on

parallelization. The examples and the case study are parallelized using the guidelines that have been introduced in the previous sections. Experiments are performed using a four Intel 2.6 GHz Xeon processor system running under a Linux 2.6 kernel. C++ is used as implementation language since it provides a basic API for the *pthread* library which allows to bind the execution of threads on a CPU basis.

The *pthread* library allows thread distribution via the `sched_setaffinity` and `CPU_SET` functions. By passing a bit mask to these methods, developer can specify the processing elements for the calling thread to run. If multiple bits of the mask are set, the operating system schedules the thread among the selected processing elements. By using *pthread* functions, an object's method can be programmed as a thread and can be bound to a processing element allowing to explicitly program distribution schemes for the objects.

Our usage of *pthread* library and `sched_setaffinity` function on a sequential implementation of the `notifyObservers()` method in the Observer pattern given in Figure 3.4. In the sequential implementation all the observers (denoted as `observers`) registered to the subject are notified in a loop sequentially.

```
1 void Subject::notifyObservers(){
2   for(int i=0;i<numOfObs;i++)
3     observers[i]->notify(this->state);
4 }
```

Figure 3.4: Sequential implementation of `notifyObservers`.

The parallelized version of the `notifyObservers()` method is shown in Figure 3.5. Instead of sequentially calling each observer's `notify()` method, a new thread is created for each `notify()` call (at line 8) and provided with thread specific information using `observerData`. In this data structure, the observer object that is going to be updated (line 4), the affinity of the observer thread (line 5) and the state of the subject (line 6) are passed to the observer thread.

In this example, the processor affinities of the observer threads are determined using a round robin scheduling algorithm. Briefly, in the subject's notification loop a new thread is created for each observer passing observer specific data to the thread as parameter. The actual processor assignment and observer notification process is carried out in the thread's function (`obs_thr`) shown in Figure 3.6 which is nothing but an ordinary function that is conventionally used in `pthread` programming.

```
1 void Subject::notifyObservers(){
2
3   for(int i=0;i<numOfObs;i++){
4     observerData[i]->obs=observers[i];
5     observerData[i]->affinity=i%numOfPRoc;
6     observerData[i]->state=this->state;
7
8     if(pthread_create(&p_thread[i], NULL, obs_thr,
9                                 observerData) != 0)
10      fprintf(stderr, "Error creating the thread");
11  }
12
13  for(int i=0;i<numOfObs;i++)
14    pthread_join(p_thread[i], NULL);
15 }
```

Figure 3.5: Parallelized implementation of `notifyObservers`.

The function that is executed for each observer thread is presented in Figure 3.6. After saving the parameter that has been passed to the thread (line 2), the bit mask holding the processor affinity of the thread is set (line 5). Processor binding is performed right after the bit mask is set (line 7) and finally the observer's `notify` method is called (line 10), updating the status of the observer.

```
1 void* obs_thr(void* arg){
2   OBSERVERDATA* oData=(OBSERVERDATA*) arg;
3   cpu_set_t mask;
4
5   CPU_SET(oData->affinity, &mask );
6
7   if (sched_setaffinity(0, &mask) <0)
8     perror("sched_setaffinity");
9
10  (oData->obs)->notify(oData->state);
11 }
```

Figure 3.6: `Observer`'s update thread.

During design pattern implementation, each object in a pattern is programmed as a separate thread with a dummy workload. Many different examples of dependency patterns can be found in the case study; more notable ones are focused below. In all the plots presented below, y-axis represents normalized runtime performance.

$$\rho_i = \frac{1}{T_i} \tag{3.1}$$

$$\rho_n = \frac{\rho_i}{\rho_i^{(best)}} \times 100 \tag{3.2}$$

In Equations **(3.1)** and **(3.2)**, T_i represents avarage running time for each case, ρ_i represents performance of each case and $\rho_i^{(best)}$ is the best performance(lowest T_i,

highest ρ_i) among all measurements for the plot at hand. Multiplicating the result by 100 enables to easily read the performance differences between measurements with terms of percentage.

3.2.4.1 Dependency pattern occurences in selected design patterns

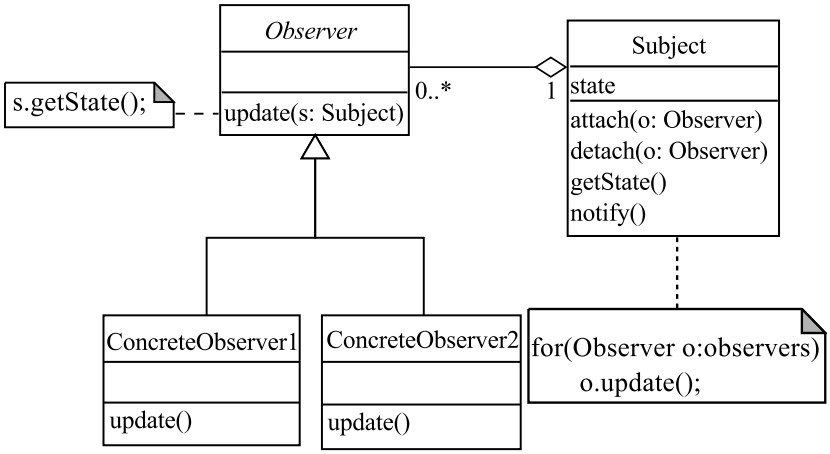
A "one authority to many sub-classes" pattern can be found in the observer design pattern shown in Figure 3.7(a). Based on the above described guidelines, subclasses of the `Observer` should be distributed and the `Subject` class should be synchronized. For this pattern, we study the effect of distributing a pool of identical workloaded observers over multiple processors, in order to see if distribution of observers improves the overall performance of the system. Each observer is configured to have an equal amount of latency when the update to the subject is posted to each observer. The time spent after a number of update operations is evaluated for different numbers of observer objects running on the utilized processors.

Figure 3.7(b) shows the performance of the parallelization for different number of processors (processing elements). Because of the read-only behavior of the pattern all processing elements are fully utilized. The plot shows how the performance increases linearly with the number of processors.

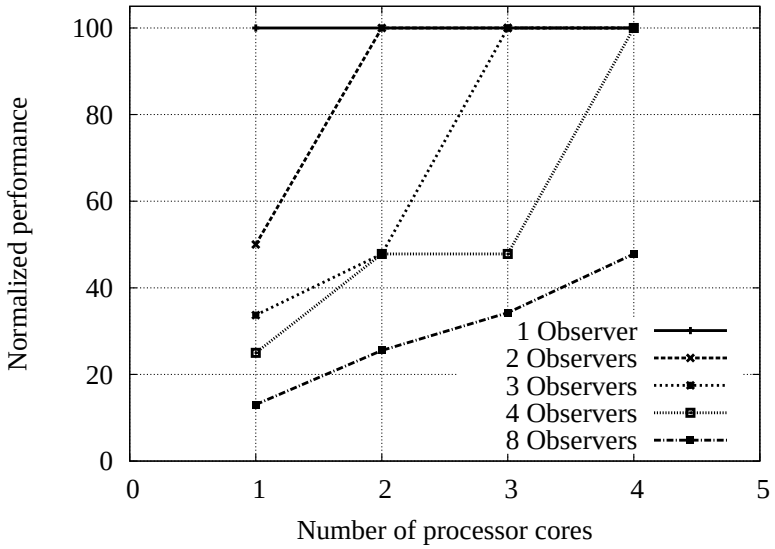
In a loop, the `Subject` updates all observers sequentially. Updating is independent for each observer, and therefore can be performed in parallel. The `state` variable should be synchronized since race conditions can occur during the state change of the `Subject`. An observer can obtain an inaccurate value if the `Subject` tries to change its state during the update. [62] contains a more detailed discussion of the experiments with the observer pattern.

The decorator pattern shown in Figure 3.8(a) exhibits self-dependency. This is handled by decomposing the class introducing a "has-a" relation. `ConcreteComponent` is decomposed into many `SubComponents` and access to each `SubComponent` by the decorator is parallelized.

Parallelization of the decorator pattern is performed in a way so that each decorator operates in parallel on a different `SubComponent` object and only one decorator can operate on a `SubComponent` object at a time, making `SubComponent` objects critical sections. Instead of using only one critical section (the `ConcreteComponent`), certain



(a) Observer Design Pattern.

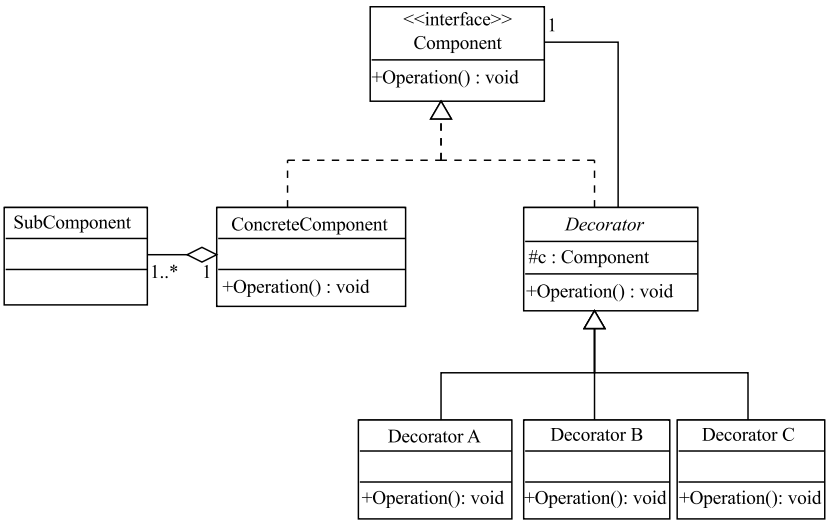


(b) Observer Parallelization Performance.

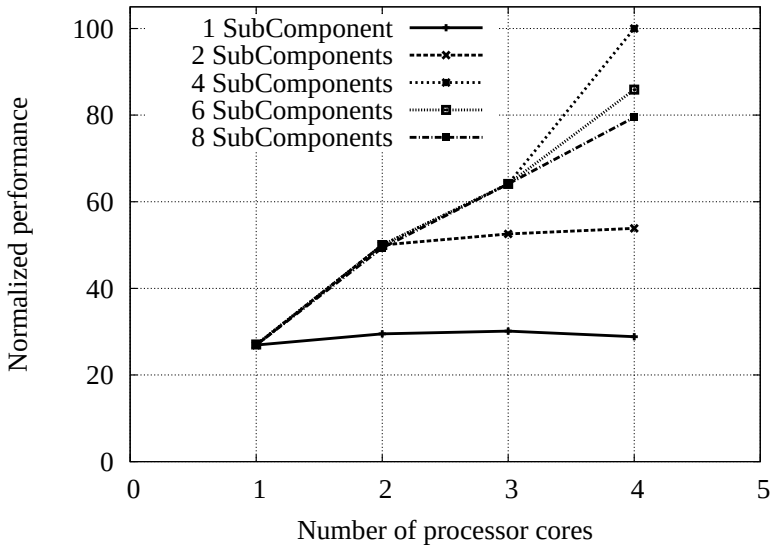
Figure 3.7: Parallelization of observer.

elements of the class are decomposed and isolated into many different classes called `SubComponents` that act as separate critical sections. The self-dependency pattern can be used in spotting such opportunities for parallelism.

The performance results for the parallelized decorator pattern are shown in Figure 3.8(b). The speed-up continues until the number of subcomponents reaches the number of processors (processing elements). Also, if the number of components goes beyond the number of processor, performance degrades. There exists a natural bound on the number of sub-components as they must be protected as critical sections. Here the dependency pattern enables the software developer to make implementation decisions as to how many threads to employ based on observing that the relation



(a) Decorator Design Pattern.

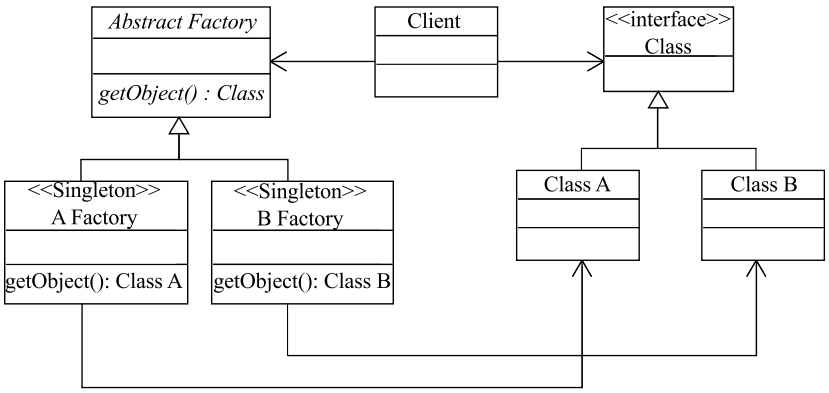


(b) Decorator Parallelization Performance.

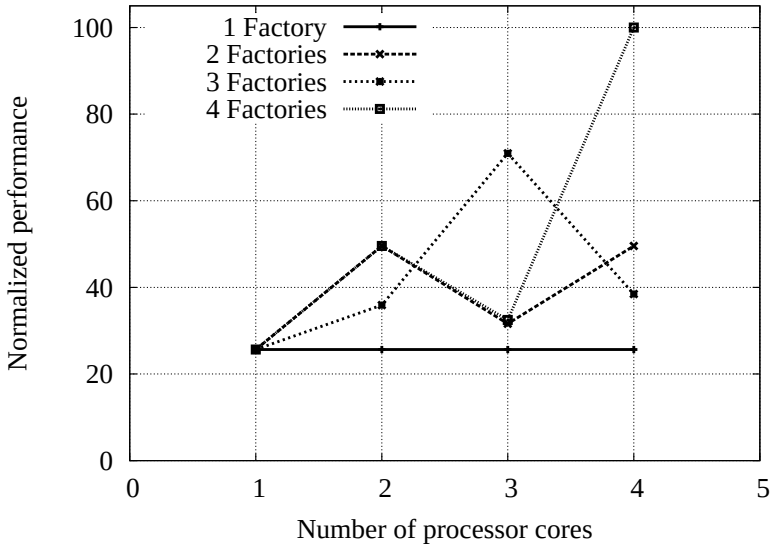
Figure 3.8: Parallelization of decorator.

between the number of `SubComponents` and the number of processing elements affects performance. [63] provides additional information concering these experiments for the decorator pattern.

In the abstract factory pattern shown in Figure 3.9(a), a “flow bridge” is present where a client is dependent on each concrete factory and each concrete factory is dependent on the interface named `Class`. Figure 3.9(b) shows the performance of our implementation when factory objects are distributed among processors (processing elements) manually in an ordered fashion. Each type of factory object is responsible of creating objects of the same type. (Object creation is represented by a specific



(a) Abstract Factory Design Pattern.



(b) Abstract Factory Parallelization Performance.

Figure 3.9: Parallelization of abstract factory.

workload value where a factory spends a predetermined amount of time during object creation.)

During the experiments with factories, when the number of processors is equal to the number of concrete factories, a concrete factory is always assigned to the same processor which means that each processor always creates only one kind of object. In a less balanced distribution, processors may produce different kinds of objects each time it is necessary to create an object. This results in different processors waiting for each other in order to gain access to the singleton concrete factories. For a two system with two processing elements, optimal number of concrete factories are multiples of two while for a three processor system this number becomes multiples of three and for a four processor system it becomes multiples of four. It is important here to remember

object distribution is made with standard scheduling algorithm where each object that is requested to be created, its creator factory is assigned to the processor in a round robin way, more formally like in Equation **(3.3)**.

$$\texttt{affinity}(O_i) = i \mod k \quad \textbf{(3.3)}$$

$\texttt{affinity}(O_i)$ is function that sets the affinity for each object to be created in the system and k is the number of processing elements available. It is assumed that each kind of object is requested to be created sequentially. Looking at our distribution scheme it is natural to have this as a result. When the number of processors is equal to the number of concrete factories, a concrete factory is always assigned to a processor which means that processor always creates one kind of object. In a less balanced distribution where number of processors is not a factor of number of types of concrete factories, each processors may produce a different kind of object each time. This results in different processors waiting for each other in order to gain access to the Singleton concrete factory. This result is quite important because we see the importance of locality in a parallel environment where keeping a constant concrete factory in each processor increases the performance more than simple parallelization.

Here the importance of locality in a parallel environment can be seen where keeping the same concrete factory in a processor increases the performance more than randomly distributing factory objects. This leads to the principle that frequently accessed synchronized objects of a flow bridge should be bound to specific processors instead of being migrated often. This result conforms with the guidelines proposed in Section 3.2.3.

3.2.4.2 Dependency patterns in a real-world software

The Jikes case study exhibits many examples of dependency patterns. The class `Control` is a "master class" shown in Figures 3.10 and 3.11. In this class, the main functions of the compilation process, such as lexical analysis (scanner), syntactic and semantic analysis (parser), and code generation, are triggered. Further, if several files are compiled, these separate compilations are handled *sequentially* by loops in the `Control` class. By parallelizing the `Control` class detected by dependency pattern analysis the compilation process can be parallelized. The compilation process involves

many independent operations (such as syntactic analysis of each file) that can be performed in parallel when compiling multiple files.

After injecting parallelism into detected regions, a performance improvement can be seen for multi-file compilation as shown in Figure 3.12. The performance results for different numbers of processors are obtained compiling similarly sized files and compiling files with very different sizes. After parallelizing the above loops, instances of `Scanner`, `Parser`, and `StoragePool` are executed in each thread. The dependency diagram for `Control` (see Figure 3.11) reveals a dependency relation from these classes to `Control`. Therefore, all of the dependent classes of `Control` become candidates as thread parameters.

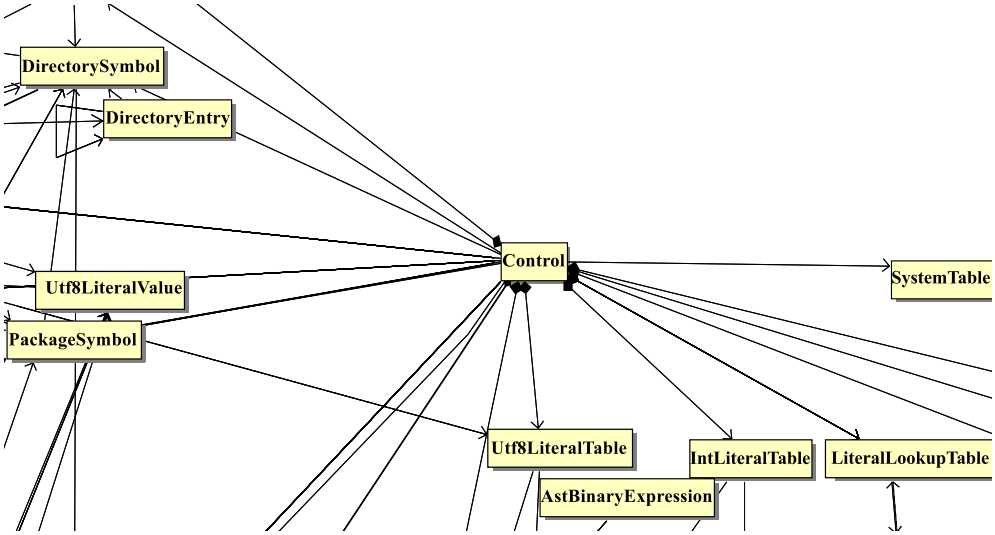


Figure 3.10: `Control` as a master class (also see `Control`'s complete dependency diagram in Figure A.2).

As a compiler has an inherently sequential nature, parallelizing the independent file compilation process is one of the more beneficial optimizations. Naturally, performance improvement for file compilation is not as great as for the small examples of the design patterns: a performance improvement of approximately 10% can be seen in Figure 3.12. This performance improvement is obtained without any detailed insight into the software being parallelized, merely by parallelizing a few loops found in inspecting the class identified as a dependency pattern. With additional insight into the software and applying parallelization at the initial implementation of the system, further improvement in performance might be obtained.

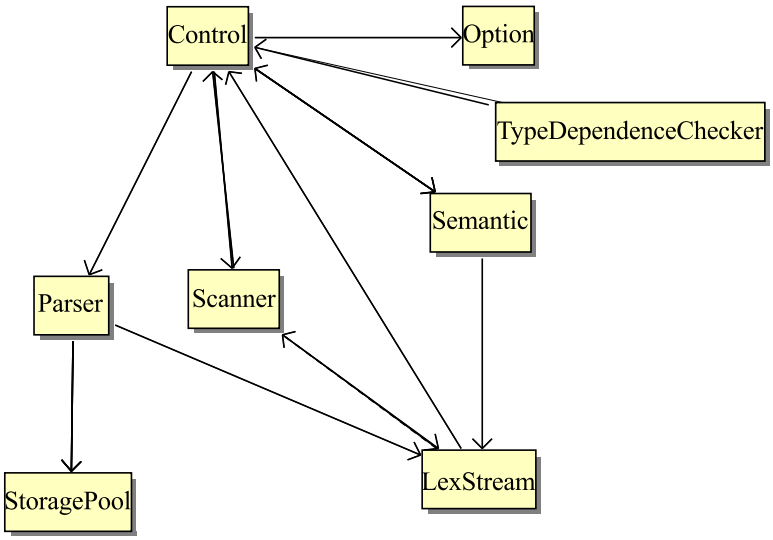
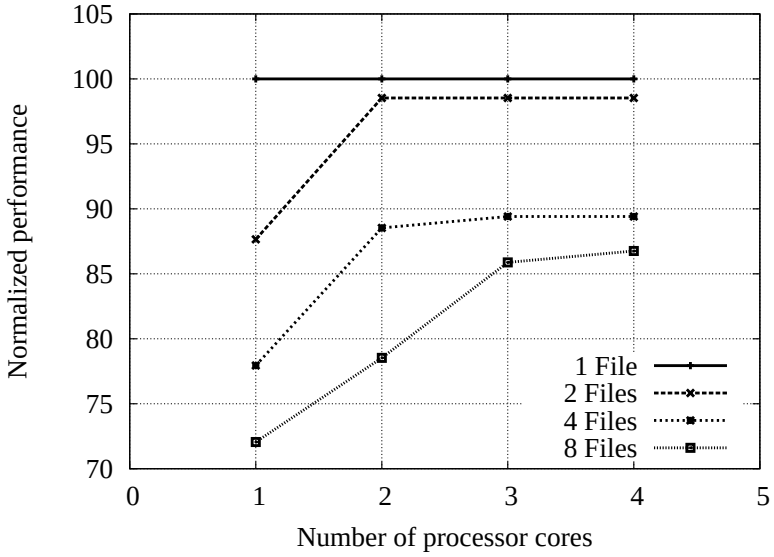


Figure 3.11: Dependency relations of `Control`.

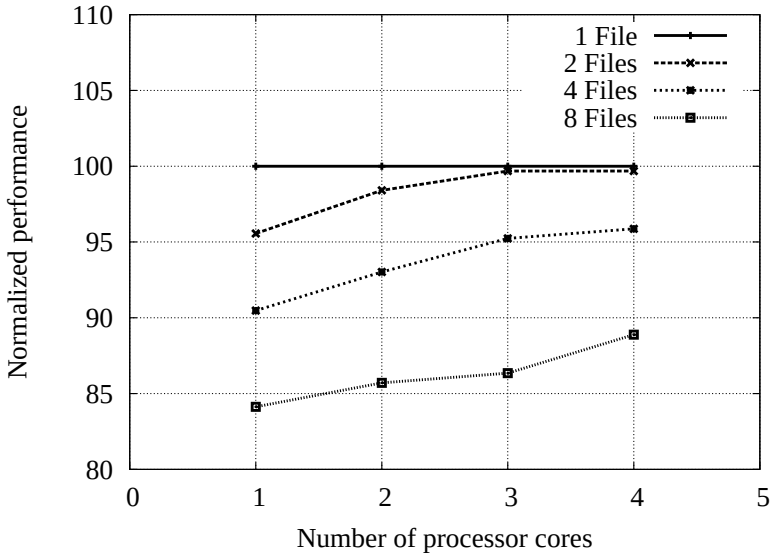
For parallelization it is not sufficient to solely discover parallelizable sections. Important locales in the software that need to be protected against race conditions caused by parallel access should be discovered as well. The first locales to look for such conditions are regions of the design where dependency relations are concentrated. Those regions may experience interactions of the many relations they hold to other classes. Dependency patterns allow us to locate such potentially interacting code sections without requiring runtime information. An authority superclass is not only a frequently accessed part of the software by being an authority, but also encapsulates common information among its descendants by being a superclass. Consequentially, an authority superclass will often be used frequently at runtime and may require synchronization when parallelizing.

The class `AstExpression` from the case study illustrates the properties of an authority superclass as shown in Figure 3.13. `AstExpression` is a generalization that represents the various nodes of the syntax tree. Due to the dense dependency relations of the superclass, these nodes will be frequently referenced in a polymorphic way. The sub-classes differ only with respect to a small set of their properties but have much in common. When the compiler is parallelized they should share synchronization properties and `AstExpression` as their superclass will be a good place to handle this synchronization.

For example, other than its constructors, `AstExpression` has only one base class method, `IsConstant()`, which is responsible to check the value of a public instance



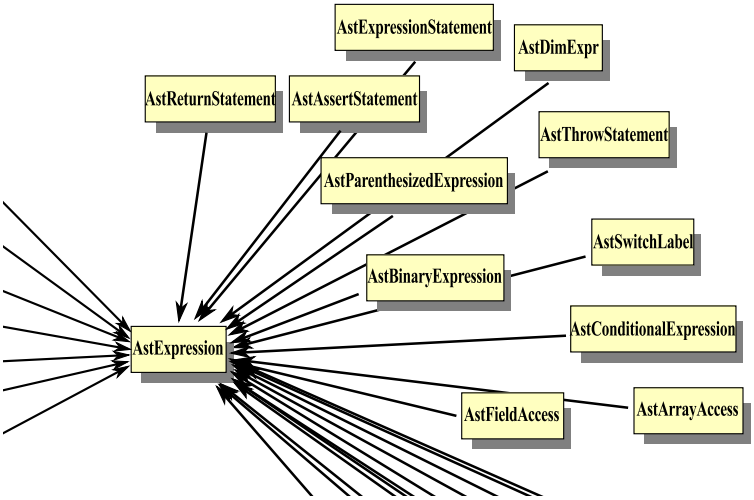
(a) Compilation of uniform length files.



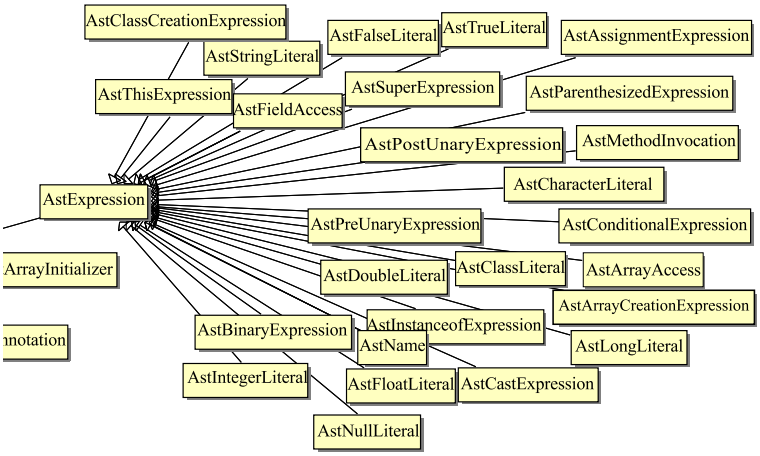
(b) Compilation of varying length files.

Figure 3.12: Jikes performance improvement by master class parallelization.

variable. Although this method is a one-liner, it is called often during compilation. For example, when compiling 400 lines of Java code, this method is called from 730 different objects, where each object calls this function approximately 5 times. This heavy access traffic to the method makes it a potential critical section. Paying extra attention to synchronization points such as in this example will prevent race conditions that may occur in instance variables of `AstExpression`.



(a) `AstExpression` Dependencies (also see `AstExpression`'s complete dependency diagram in Figure A.3).

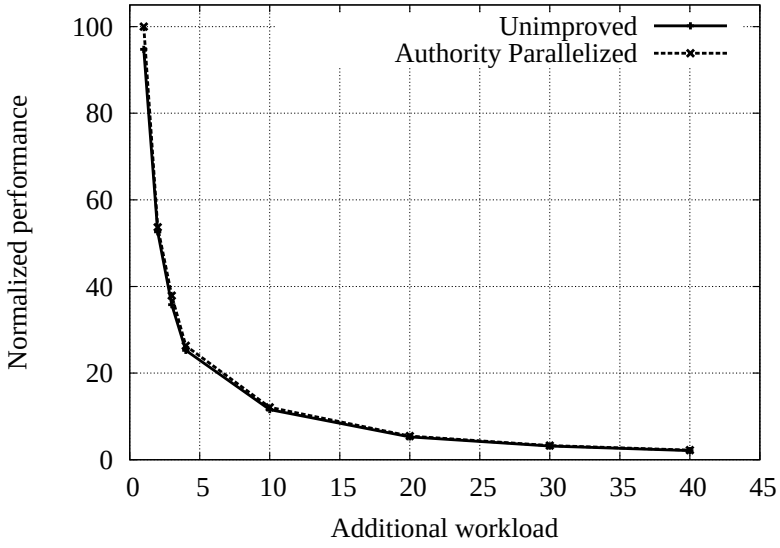


(b) `AstExpression` inheritance relations.

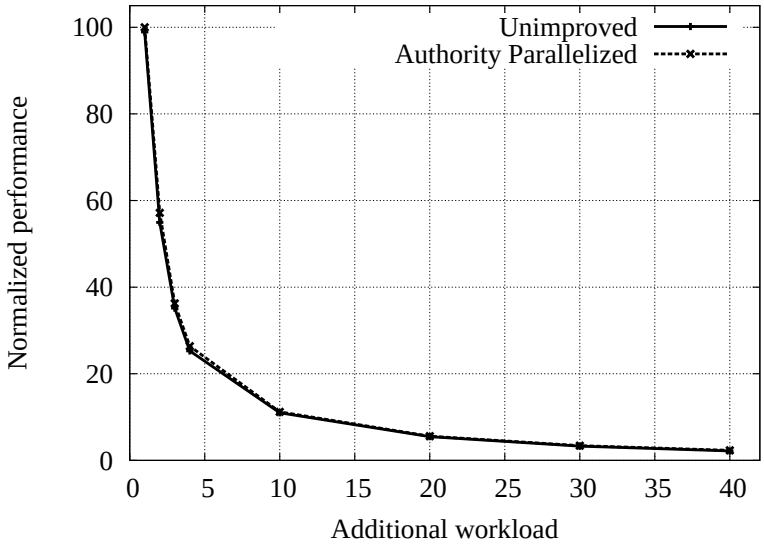
Figure 3.13: `AstExpression` as an authority superclass.

Parallelism is injected to those regions where authority superclass objects are accessed frequently. Figure 3.14 shows a small performance speed-up in a four processor system as the workload of the authority superclass increases.

An example of a bridge pattern can be seen in Figure 3.15 (the complete dependency diagram of `CLASS` is shown in Figure A.1), where a subset of the descendants of `Attribute` form an authority bridge. In the implementation of the classes at the ends of the bridge, indicated by S in Figure 3.15(a), bridge objects are used in a similar alternating way when they are accessed in loops. `ClassFile`, `MethodInfo` and `FieldInfo` maintain instances of subclasses of `AttributeInfo` in a buffer array



(a) Single-file compilation performance.

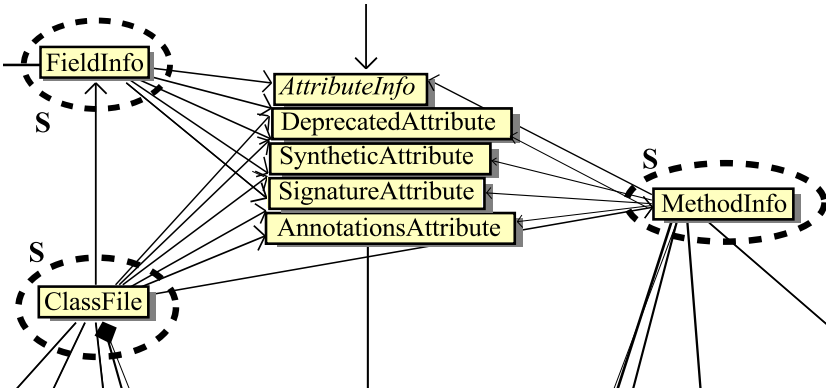


(b) Multi-file compilation performance.

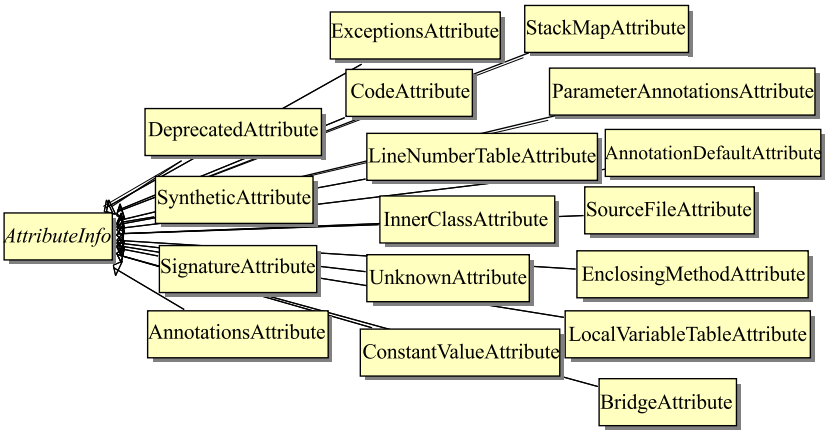
Figure 3.14: Parallelization of authority superclass.

which is then iterated over in the mentioned loops. This situation needs special care while distributing and recombining the buffer of the bridge objects.

A bridge object access can be found in the constructor for above classes. The constructor method contains a switch statement in which appropriate actions are taken depending on the attribute type. This switch is executed many times in a loop for each `AttributeInfo` object. By parallelizing this loop the performance improvement shown in Figure 3.16 is obtained, as the workload of the operation increases in a four processor system.



(a) `AttributeInfo` Dependencies (also see `AttributeInfo`'s complete dependency diagram in Figure A.1).

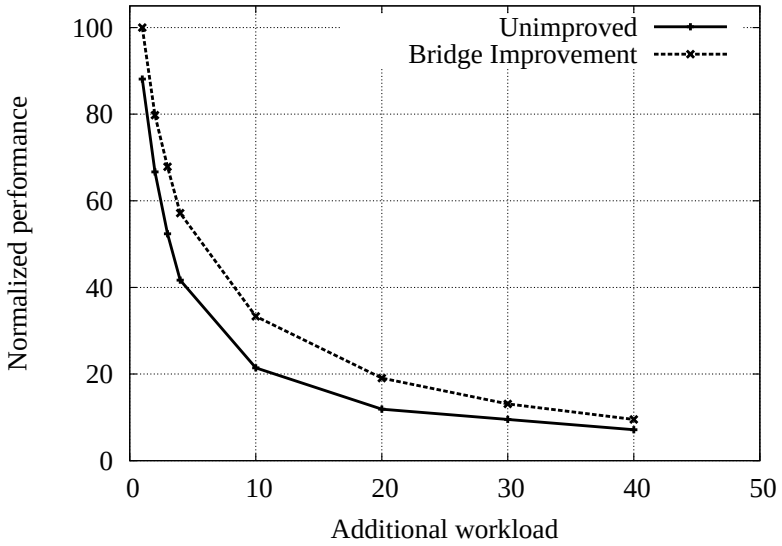


(b) `AttributeInfo` inheritance relations.

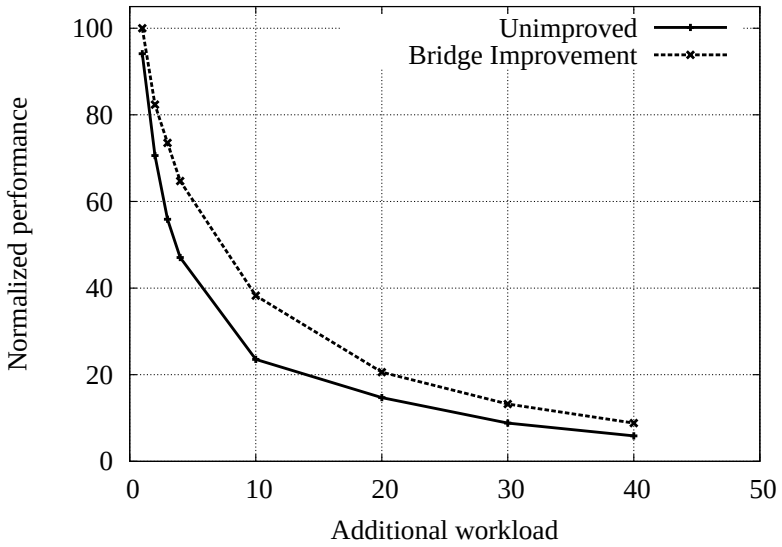
Figure 3.15: `AttributeInfo` descendent's as a hub bridge.

Another example of a bridge can be seen in Figure A.3 where a subset of the descendants of `AstStatement` form a hub bridge between `AstBlock` and `AstExpression`. All classes that are part of the bridge are statements which are excessively processed inside a method of the bytecode generator class. This method contains a huge switch statement in which appropriate actions are taken depending on the statement type. This method is called in a loop, successively processing each statement based on a polymorphic parameter. When parallelizing this loop, the two ends of the bridge(`AstStatement` and `AstExpression`) become important as they need to be protected against parallel access. The findings for `AstExpression` as an authority superclass also coincide with the role of this class in the bridge pattern.

Dependency pattern analysis provides two different advantages when analyzing a software design. The first advantage is the possible performance improvement by pinpointing opportunities for parallelization. The second advantage is the identification



(a) Single-file compilation performance.



(b) Multi-file compilation performance.

Figure 3.16: Parallelization of authority bridge. The horizontal axis indicates the additional workload handled by the parallelized system: A value of "10" means that each bridge class handles a 10 times larger workload than in the unparallelized version.

of inherently sequential regions which have the potential to cause bottlenecks for system performance. Developers can leverage workload by loading the inherently sequential regions as lightly as possible and shifting the workload to the parallelizable regions. For example, in a web browser, replicating the address and navigation bars in each tab can be a good parallelization opportunity and minimize a potential bottle neck. It is better to have bridge classes with large workloads rather than concentrating the workload in authority superclasses.

3.3 A Metric Set for Dependency Patterns

Software metrics are the means of measurement that are becoming increasingly popular for modern object oriented software. Metrics can be used to measure some property of a piece of software or its specifications. Software metrics are not specific to object oriented software or their application area is not specific to programming stage; there also exist metrics for imperative software or metrics for software design. Metric usage address to make estimations on various aspects of software like robustness, maintainability and reusability.

Measuring software properties is an important and yet a vague area of software engineering. Many different metrics have been proposed through time considering different properties of software. What makes the area vague is that it has never been possible to completely define all the attributes that a specific metric represent. Moreover, it is very hard to empirically validate an exact recipe using metrics that increase the software quality. Software quality being a subjective and multi-dimensional concept, is the main reason behind these difficulties. In order to define metrics serving their cause as much as possible, it is important to precisely define their application domain and the attributes of software that they measure. During the thesis studies, dependency patterns are chosen as the application domain of the metrics to be defined.

In this section metric definitions to conduct a finer analysis on individual properties of dependency patterns and their place in object oriented software are presented. A finer analysis for dependency patterns is needed because of many reasons. Some of those reasons can be listed as follows:

- A distinction/quantification is needed among occurrences of individual patterns types inside class diagrams according to the different properties they have. When they are detected based on their general definitions a crisp distinction between pattern types can be obtained. However by measuring detailed properties using metrics a more continuous distinction among dependency patterns can be obtained.
- In some cases, arbitrary classes/class groups can contain dependencies coinciding with specific dependency patterns. It is needed to sort out those false alarms by analyzing their specific properties deeper and having a deeper insight about their role in class diagrams.
- Current metrics in the literature are not defined to measure specific properties of dependency patterns and needs to be tailored(and new metrics need to be defined) for dependency patterns.

Enriching the specification power of dependency patterns using design metrics can provide a stronger connection between static software design and runtime behavior of the software. This will allow the designer to gain a better foresight on implementation stage of software. By defining dependency pattern specific metrics it will be possible to relate them with software parallelization concepts providing recipes based on metric values. When the dependency patterns are used as a connection between static and dynamic properties of software they can provide a basis for establishing a connection between software design metrics and parallelization as well. Another advantage of using dependency pattern based metrics is obtaining groups of classes formed by a particular grouping strategy defined by dependency patterns. Previous studies on design metrics for groups of classes only use software packages as subjects.

3.3.1 Related work on pattern metrics

Through the history of object oriented software development, metrics have been an important mean of measurement in evaluating quality of different software aspects. It is not possible to define a single recipe for assessing quality because it has many different dimensions as a concept. For this reason the process of metric derivation becomes more effective if a metric at hand is shown to be theoretically valid in measuring properties of software which it was designed to assess.

Quality assurance methods becomes more effective when they are applied at initial phases of system implementation. Design phase is one of these early stages where pictures of the software are drawn from different perspectives. Chidamber and Kemerer carried out pioneering work [64] in the field of software design metrics, which have been used as a touchstone in many of succeeding studies. For instance Harrison et al. compared MOOD metric set [65] with Chidamber's metrics to show that two sets are complementary and offer different assessments of a system [66]. Later Bansiya and Davis extended this metric set [67] to build a hierarchical method for object oriented quality assessment. Briand et al. also made important studies especially about the coupling metrics [68, 69] of object oriented software design.

For the field of parallel software, performance is the primary concern leading the field towards developing performance metrics. One of the earliest examples is DePaoli and Morasca's work on adopting complexity metrics, like McCabe's cyclomatic complexity [70], to concurrent Ada software. Many other performance models/metrics exist in literature like [71] in which resource metrics are used to characterize the various models of parallel computation. Another example is Hollingsworth and Miller utilizing existing performance metrics in a new technique which they call "True Zeroing" [72]. Parallelization metrics are defined at a lower level compared to object oriented design metrics and in the last decade a few studies exist that relate the two distinct fields. One of these studies is by [73], where they describe how to measure and attribute arbitrary performance metrics for a high-level multithreaded programming model known as Cilk [74].

The relation between software design metrics and design patterns are also another field of study where most of the research is being done on detecting design patterns using design metrics. And et al. conducted a study on this subject [75] where they feed a multi-stage reduction strategy based approach with object oriented software metrics to extract structural design patterns from software design/code. Another study [76] use metrics to measure the improvement when software design patterns are used in software development. Lastly, Robert Martin described a set of dependency metrics that measure the conformance of a design to the desirable pattern [77].

Almost all of the examples above are class centric methods or quality assessment models based on class metrics where relations among groups of classes are ignored

most of the time for a simple reason: lacking a stable grouping principle. Robert C. Martin's software package metrics [78] satisfies this shortage by defining the grouping principle as software packages, hence allow to infer about dependencies among classes vastly. However dependencies are not specific to inter-package relationships; they also exist among classes in a package. On the other hand, it is important to remember again a stable grouping principle is needed, in order to define and apply metrics over groups of classes and relationship among those.

Following all the information above, a missing piece of the puzzle can be found out in relationships among software design metrics, parallel software, design patterns and multi-class dependency metrics. Dependency patterns, stand at a place between parallel software and object oriented software design. They fill the gap between the expression of logical concurrency in software and its realization at run-time. Software design can be assessed regarding parallelization using dependency patterns by using the proposed metrics and effects of these properties on the software implementation.

3.3.2 Dependency pattern specific metrics

In this section a set of metrics is proposed for each type of dependency patterns introduced in the last section. Metrics are exemplified using simple examples and interpretations of the possible metric values are explained. After metric definitions, a study on correlation among the metrics is also presented, showing that the metrics cover different attributes of patterns. It is important to cite that some of the metrics below are adapted to dependency patterns from software package metrics [78].

3.3.2.1 Hub/Authority metrics

Ratio of Dependency Directions

Ratio of Dependency Directions(RDD) of a class measures dominance of its afferent/efferent dependencies using the ratio of the difference between its afferent and efferent dependencies over its total number of dependencies. More formally:

$$\mathtt{RDD} = \frac{D_{out} - D_{in}}{D_{tot}} \tag{3.4}$$

In Equation **(3.4)**, D_{out} represents the number of direct dependencies of the class towards other classes while D_{in} represents number of direct dependencies to the class

and D_{tot} represents total number of dependencies that the class have. For instance RDD will be calculated as 0.2 ((3-2)/5) for `Class S` in Figure 3.17

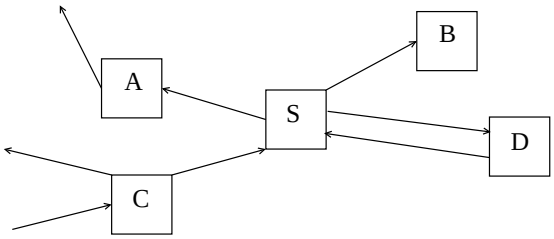


Figure 3.17: An example class for hub/authority metrics.

RDD defines the amount of hubness/authorityness of a class; if the metric values is close to 1, class at hand shows hub properties and if metric value is close to -1 the class shows authority properties. This metric should be applied after a class has been selected as a hub/authority. For instance a class with only two efferent dependencies can be said to have hub properties using this metric. However it has too few dependencies to be identified as a hub or an authority.

In terms of parallelization, having a RDD closer to 0 indicates a higher parallelization effort. In this case, class' afferent and efferent usage is balanced which brings out lots of possible dependency conflicts in software. On the other hand if the metric is closer to 1, parts of class showing hub properties can be isolated easier, making the class suitable for introducing parallelization. When the metric is closer to -1, this is an indication of a heavier synchronization work since class is mostly used by other classes.

Ratio of Singular Dependencies

Ratio of Singular Dependencies(RSD) of a class measures dominance of singly dependent class dependencies to/from the subject class. More formally:

$$\texttt{RSD} = \frac{D_{sng}}{D_{tot}} \tag{3.5}$$

In Equation **(3.5)**, D_{sng} represents the number of direct dependencies that has been solely made to the subject class and D_{tot} represents total number of dependencies that the subject class have. For instance RSD will be calculated as 0.6 (3/5) for the class in Figure 3.17 since `Class B` and `Class D` has dependencies only to `Class S` not to another class while `Class A` and `Class C` has other dependencies as well.

RSD defines the amount of independence of a class and its dependents as a whole; if the metric values is close to 1 class at hand and the classes that have dependency relationships with it can be handled more independently from the rest of the system. Also if the classes that have their only relationship with the subject class are also the children of the subject class, this can be a good indication of polymorphic usage.

In terms of parallelization, having a RSD closer to 1 indicates an easier parallelization process since the developer would only be concerned about singular dependencies towards/from the class at hand.

3.3.2.2 Cycle metrics

Number of Cyclic Dependencies

Number of Cyclic Dependencies(NCD) of a class measures the number of dependencies that a class has towards itself. For instance NCD will be calculated as 2 for the `Class S` in Figure 3.18.

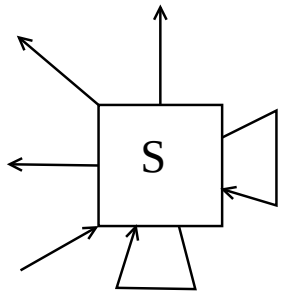


Figure 3.18: An example class for cycle metrics.

Having a high value of NCD indicates more effort on parallelization. However the outcome of this effort can be predicted with an additional metric which is defined next.

Ratio of Cyclic Dependencies

Ratio of Cyclic Dependencies(RCD) of a class measures dominance of its cyclic dependencies by measuring the ratio of its self dependencies over its total number of dependencies. More formally:

$$\texttt{RCD} = \frac{D_{cyc}}{D_{tot}} \tag{3.6}$$

In Equation **(3.6)**, D_{cyc} represents the number of self dependencies of the class and D_{tot} represents total number of dependencies that the class have. For instance RDD will be calculated as 0.33 (2/6) for the `Class S` in Figure 3.18.

RCD defines the amount of self dependency of a class; if the metric value is close to 1, class' whole purpose becomes based on the cyclic dependency. In terms of parallelization, number of cyclic dependencies should be considered before analyzing this metric. If the class has many cyclic dependencies and the value of RCD is also high, developer should pay more attention on resolving and parallelizing self dependencies inside the class. However in this situation, the subject class usage is a bigger threat to the parallelization since it has many cyclic dependencies scattered around the software, sequentializing software run.

When both of the metrics are low, there are many dependencies towards a single cyclic dependency which holds a potential for a performance boost when small numbers of self dependencies are resolved. When RCD is low and NCD is high resolving many self dependencies may end up with a local performance boost. Finally if RCD is high when NCD is low a small effort may provide a local performance boost.

3.3.2.3 Bridge metrics

Ratio of External/Internal Bridge Dependencies

Ratio of External/Internal Bridge Dependencies(REIBD) of a bridge measures dominance of its internal/external dependencies using the ratio of the difference between its external and internal dependencies(excluding the source connections) over its total number of dependencies. More formally:

$$\texttt{REIBD} = \frac{D_{ext} - D_{int}}{D_{ext} + D_{int}} \quad \textbf{(3.7)}$$

In Equation **(3.7)**, D_{ext} represents the total number of direct dependencies that has been made towards/from outside the bridge(excluding source connections) and D_{int} represents number of direct dependencies bridge classes make among themselves. For instance REIBD will be calculated as 0 ((3-3)/(3+3)) for the bridge in Figure 3.19. In the figure S1 and S2 are source classes of the bridge.

In a bridge, absence of external or internal dependencies is frequent so it is not practical to use a simpler formula like D_{ext}/D_{int} to measure to dominance of dependencies.

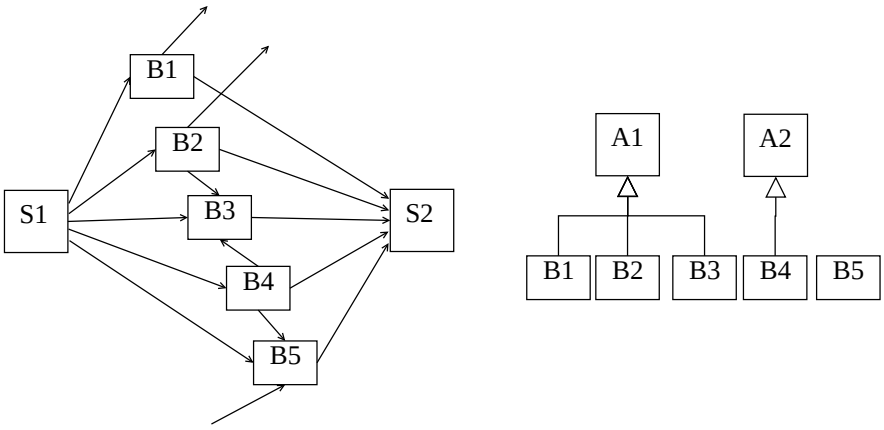


Figure 3.19: An example for bridge metrics.

REIBD defines the amount of independence of a bridge; if the metric values is close to -1, bridge at hand mainly has dependency relations with the classes out of the pattern and if metric value is close to 1 the bridge classes are mainly dependent among each other. This metric only shows dominance of internal/external bridge dependencies.

In terms of parallelization, having a REIBD closer to 0 indicates a harder parallelizability, since it shows that the bridge has same amount of internal and external dependencies. In practice having a value closer to -1 is better since bridges may be isolated easier if they don't have any dependencies outside the pattern. The metric is not defined for the bridges that doesn't have any dependencies apart from its source dependencies.

Ratio of Bridge to Source Dependencies

Ratio of Bridge to Source Dependencies(RBSD) of a bridge measures purity of bridge dependencies using the ratio of its source dependencies over its total number of dependencies. More formally:

$$\texttt{RBSD} = \frac{D_{src}}{D_{tot}} \tag{3.8}$$

In Equation **(3.8)**, D_{src} represents the total number of direct dependencies of bridge classes to/from source classes and D_{tot} represents total number of dependencies that bridge classes have. For instance RBSD will be calculated as 0.625 (10/16) for the bridge in Figure 3.19

RBSD defines the amount of dedication of bridge classes to the pattern; if the metric value is close to 1, bridge at hand mainly has dependency relations with the sources of

the bridge and if metric value is close to 0 the bridge classes has more dependencies other than its sources. This metric should not be too close to 0 for a bridge since the bridge loses most of its properties when it has more non-source dependencies.

In terms of parallelization, having a REIBD closer to 1 indicates easier parallelization for a bridge. Developer wouldn't have to deal with unrelated dependency relations when parallelizing the bridge, facilitating bridge parallelization. Also during the runtime, non-source dependencies may indicate barriers on the alternating routes of the bridge decelerating parallel behavior.

Ratio of Sibling Bridge Classes

Ratio of Sibling Bridge Classes(RSBC) of a bridge measures the density of sibling classes inside a bridge using the ratio of ancestor classes of bridge classes to the total number of classes inside the bridge. More formally:

$$\texttt{RSBC} = \frac{N_{par}}{N_{bdg}} \tag{3.9}$$

In Equation **(3.9)**, N_{par} represents number of different parents that bridge classes have and N_{bdg} represents total number of classes inside the bridge. For instance RSBC will be calculated as 0.4 (2/5) for the bridge in Figure 3.19.

RSBC actually has two dimensions. Metric can be closer to 1 when all the classes inside the bridge have separate ancestors or when there exists a few classes inside the pattern. In both cases parallelization process is relatively harder. Having many sibling classes in large bridges alternating heavily provides a better parallelization opportunity.

3.3.2.4 Island metrics

Ratio of External/Internal Island Dependencies

Ratio of External/Internal Island Dependencies(REIID) of a bridge measures dominance of its external dependencies using the ratio of pattern's external dependencies over its internal dependencies. More formally:

$$\texttt{REIID} = \frac{D_{ext}}{D_{int}} \tag{3.10}$$

In Equation **(3.10)**, D_{ext} represents the total number of direct dependencies that has been made towards/from outside the island and D_{int} represents number of direct

dependencies island classes make among themselves. For instance REIID will be calculated as 0.45 (5/11) for the bridge in Figure 3.20.

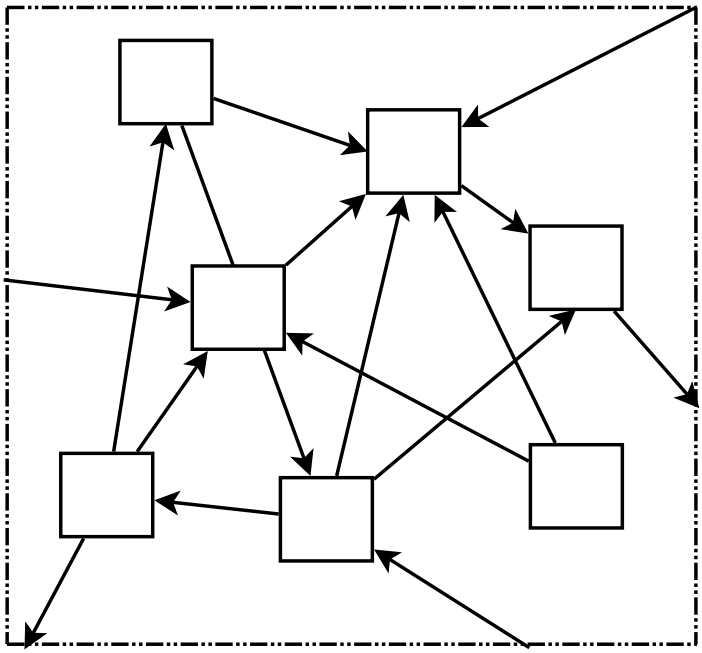


Figure 3.20: An example for island metrics.

REIID is actually used to measure the same properties of a group of classes that REIBD measures. However, by definition islands always have large number of inner dependencies which makes it viable to use a simpler ratio than REIBD. Moreover, based on the definition islands always have much more inner dependencies than outer dependencies. It is more practical to use a metric that performs a finer measurement of external dependency dominance.

REIID defines the amount of independence of an island; if the metric value is close to 0, island at hand has less dependencies to/from classes outside. This shows its independence from the rest of the diagram making the group a candidate for parallelization as a whole.

As mentioned before, classes inside the island communicate more with each other rather than the rest of the software and hence objects of the island should be placed closer among the processing elements to minimize communication cost. As the metric value increase the group starts to lose its island character. When the islands are detected prior to the metric assessment, the value of this metric should not be far from 0 in practice.

Cumulation of Inner Island Dependencies

Cumulation of Inner Island Dependencies(CIID) measures the distribution amount of the inner dependencies of an island. It can be defined as the standard deviation of number of dependencies each class has to/from other classes inside or outside the group. For instance CIID will be calculated as 1.35 (σ([3 4 4 5 2 6 3])) for the bridge in Figure 3.20

Having a small CIID shows that the dependencies of the island are distributed in a balanced way; it is harder to introduce parallelism inside the island. On the other hand when this metric is high, it shows that the dependencies are concentrated on a few classes. In this situation the island may be split up to smaller islands. Another idea is to introduce local parallelization to the heavy dependent classes being local hubs/authorities inside the island.

3.3.2.5 Correlation among dependency pattern metrics

It is very important to obtain distinct metrics that represent different properties of the dependency patterns. To reason about the distinctness of the metrics, in Table 3.1 correlation coefficients metrics that were measured using 130 different dependency patterns inside four different real-world software is presented. Case studies are introduced in more detail in Section 3.3.3. Each metric is compared with the metrics of the same dependency pattern in Table 3.1.

Table 3.1: Correlation among defined metrics.

		RDD	RSD	
H/A	RDD	1		
	RSD	-0.31	1	
		NCD	RCD	
Cycle	NCD	1		
	RCD	0.55	1	
		REIBD	RBSD	RSBC
Bridge	REIBD	1		
	RBSD	-0.63	1	
	RSBC	0.16	-0.2	1
		REIID	CIID	
Island	REIID	1		
	CIID	-0.02	1	

Two high correlated values in Table 3.1 are NCD-RCD and REIBD-RBSD couples. These correlations can be considered as natural since NCD is being used as a complementary metric for RCD, these two metrics are analyzed together in Section 3.3.2.2 to reason about cycle properties.

For the second couple, REIBD-RBSD, the reason behind the correlation is the sample space. REIBD metric is actually not specific to bridges, it can be applied to any group of classes like in REIID case for islands. However, when this metric is applied to the bridges number of external/internal dependencies becomes the complement of source dependencies in the bridge. REIBD-RBSD are also analyzed together in Section 3.3.2.3 like the former case although they also individually hold distinct properties of the pattern. On the other hand, especially analyzing REIBD without considering RBSD may mislead for some certain properties(like the pureness of the bridge). Rest of the metrics doesn't have an obvious correlation among themselves and can be used individually to reason about distinct properties of the patterns.

3.3.3 Real-world examples of dependency pattern metrics

In this section using real world software, examples of dependency patterns having different metric values will be given. Case studies are chosen from different areas and programming languages: Jikes [57] is the mid-sized compiler project of IBM written in C++, Leda [79] is an open source library of efficient data structures and algorithms written in C++, JBoss [80] is a well-known community driven application server written in Java and finally DSpace [81] is an open source CMS written in Java.

In the following sections, simple strategies on metric interpretation and metric priorities in parallelization process will be presented for each dependency pattern type. Following the parallelization proposals, examples of dependency patterns in the case study software will be presented and the metric measurements for the patterns will be revised using the examples.

3.3.3.1 Hub/Authority metric examples

Utilization strategy of hub/authority metrics in parallelization can be listed as follows:

1. In the reasoning process one should first consider RDD metric. It is better for RDD to be either close to 1 or -1. This provides a clearer parallelizing strategy based on

class at hand being a hub or an authority. Having an RDD closer to 0 represents complicated class behavior and a tedious parallelization process.

2. Based on the information from RDD, the value of RSD also becomes important. Having a higher RSD is always better but it becomes more important if the class at hand is an authority. This situation poses a possible polymorphic usage where parallelization can be introduced to the classes that use the class at hand.

The class `Control` shown in Figure 3.21 is a hub from Jikes, having a high RDD(1) and low RSD(0.02). Jikes being a compiler, `Control` is the orchestrating class of the process where the main operators of the compilation process, such as lexical analyzer(scanner), syntactic/semantic analyzer(parser) and code generator, are triggered. Further, if several files are compiled, a loop in this class handles these separate compilations *sequentially*. Compilation process of separate files are independent of each other and can be performed in parallel.

Indicated by its high RDD value, `Control` is a strong hub to which parallelism may be introduced in many different ways from method call parallelization to object distribution. In the examples below parallelize is done in a conventional way, by parallelizing loops. It is not mandatory to use loop parallelization in every case; one can not guarantee to find parallelizable loops in every situation. However it is a common construct in object oriented/imperative software that is easy to detect and parallelize; it will be one of the first places for a developer to look for a parallelization opportunity.

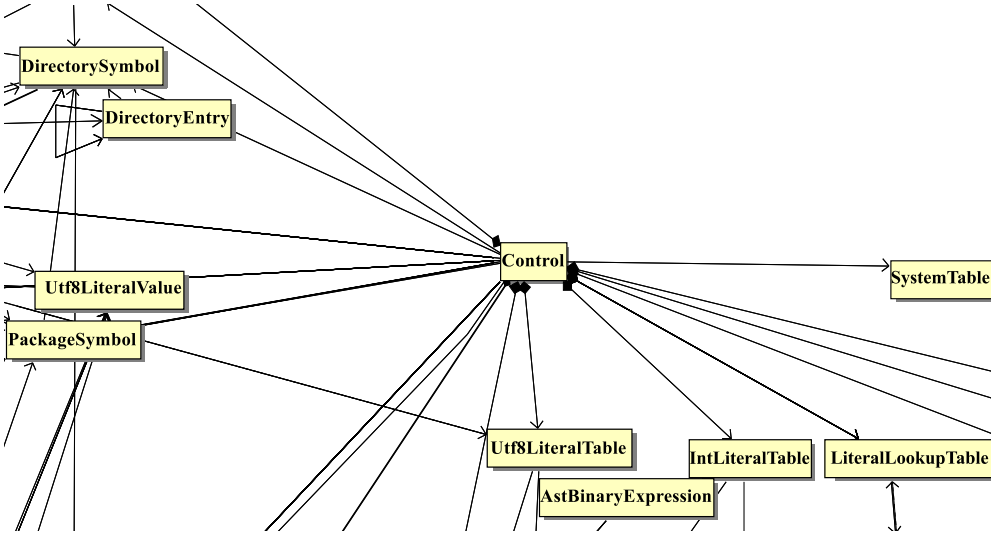


Figure 3.21: `Control` as a hub pattern.

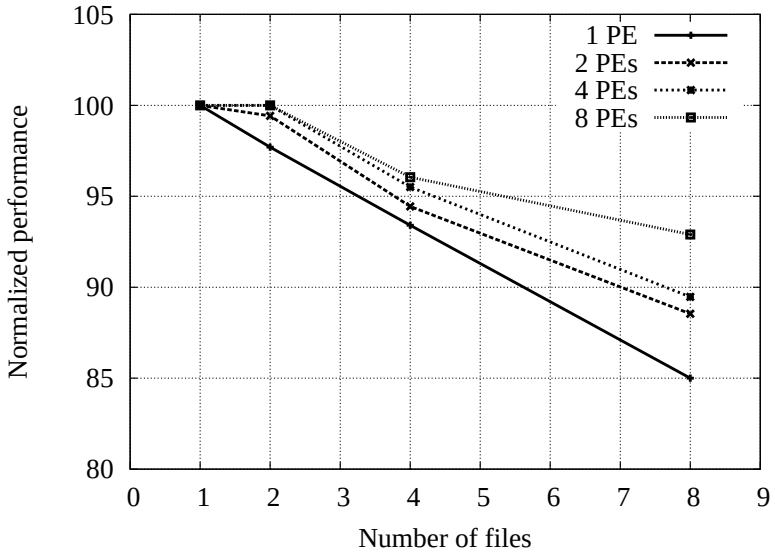
```
1 Control::Control(){
2    /* Initialization etc.*/
3    for (file_symbol = (FileSymbol*) input_java_file_set.FirstElement();
4       file_symbol;
5       file_symbol = (FileSymbol*) input_java_file_set.NextElement()) {
6         /*
7         Header Processing
8         */
9    }
10   /*Further Processing*/
11   for (int j = 0; j < num_files; j++) {
12        /*
13        Body Processing
14        */
15   }
16   /*Further Processing*/
17 }
```

Figure 3.22: Constructor of `Control` class.

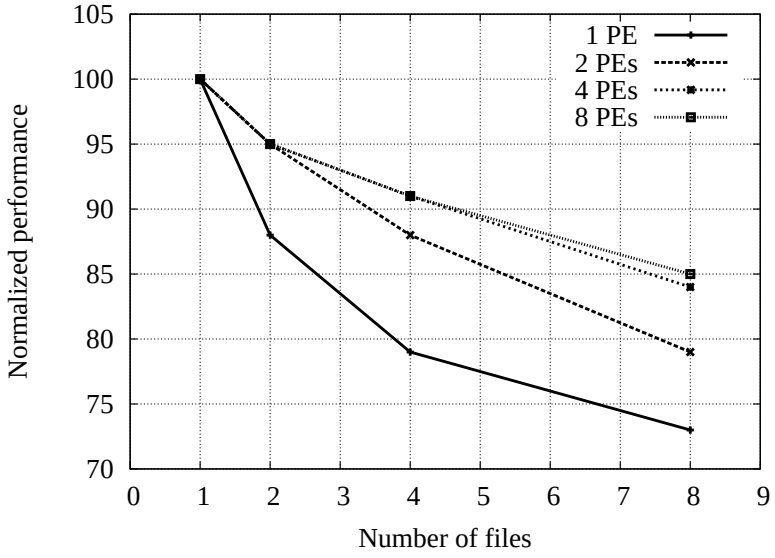
By analyzing the actual implementation of `Control` class, one may find mentioned loops which can also be seen in the code snippet in Figure 3.22. After introducing parallelism on these loops, performance improvement can be seen for multi-file compilation process in Figure 3.23. In the figure, performance numbers in (a) are obtained when identical files are compiled by the compiler using different number of processor and in (b) files with various sizes are used in compilation process. When loops in Figure 3.22 are parallelized, instances of `Scanner`, `Parser` and `StoragePool` are sent to threads as parameters. If `Control`'s dependency diagram in Figure A.3 is examined, mentioned classes can be found out to have dependency relationship with `Control`. In a more detailed parallelization process all of the dependent classes of `Control` become candidates as thread parameter. Analyzing class diagram in this way, lets the programmer to focus on important sections and classes of software before detailed code analysis.

As being a huge class `JDBCEntityBridge` from JBoss, needs a huge effort to be parallelized since its RDD(0) value indicates that it will be used as much as it will use other classes. Actually the class has 2 inline classes and 70 methods, from which 30 of them is setter/getter, 14 of them is initialization and 8 of them scheduling methods. There exists 40 different loops inside 1500 lines of codes. Consistent with the metric value, this class surely needs a lot of effort to be parallelized.

Lastly `GenPtr` is an interesting example from Leda having an RDD value of -1 and RSD value of 0.83. These values indicate that `GenPtr` doesn't actually use any other



(a) Compilation of uniform length files.



(b) Compilation of varying length files.

Figure 3.23: Jikes performance upgrade by hub Parallelization.

classes and has the potential to be used as an abstract/generic data type. In actual software, `GenPtr` is nothing but a type definition, standing for void pointers. As understood from its metric value this artifact is always being used by other classes, never explicitly using any other class.

3.3.3.2 Cycle metric examples

Utilization strategy of cycle metrics in parallelization can be listed as follows:

- **High NCD and RCD:** In this situation there are many self dependencies exists inside the class that needs more effort to break. On the other hand class is

```
1 class VariableSymbol : public Symbol, public AccessFlags{
2    public:
3    /*Various properties and methods*/
4     VariableSymbol* accessed_local;
5
6    private:
7       /*Various properties and methods*/
8 };
9
10 VariableSymbol* TypeSymbol::FindOrInsertLocalShadow(VariableSymbol* local){
11    /*Various operations*/
12
13    VariableSymbol* accessed;
14    for (accessed = variable -> accessed_local;
15          accessed && accessed != local;
16          accessed = accessed -> accessed_local);
17       assert(accessed);
18
19    return variable;
20 }
```

Figure 3.24: Self dependencies of `VariableSymbol`.

quite independent from the rest of the software, bringing local performance improvements when tweaked.

- **Low NCD and RCD:** In this situation objects of the class is being used in many places of the software and it also has a small part that obligates sequential behavior. Self dependent part of the class should be detected and analyzed to discover if it is being used heavily inside the software.
- **High NCD, low RCD:** This is one of the hardest parallelization situations where most of the class consists self dependencies and the class is being heavily used in software. This class probably becomes a bottleneck in parallelization process and should be analyzed carefully.
- **Low NCD, high RCD:** This type of self dependency is easy to detect and harmless for parallelization most of the time.

Based on the guidelines enlisted above, example interpretations of cycle metrics from case studies can be given as follows.

For `VariableSymbol` class from Jikes having an RCD value of 0.11 and an NCD value of 1 exhibits self dependency in seven different points inside software. Four of these points occur in loop conditions, causing the loop to gain a sequential behavior. One example to this situation is present in Figure 3.24.

```
1 public class SortOption{
2    /* Various Attributes*/
3    /* Self dependent attributes */
4    private static Set<SortOption> sortOptionsSet = null;
5    private static Map<Integer, SortOption> sortOptionsMap = null;
6
7    /* Various methods*/
8    /* Self dependent methods*/
9    public static Map<Integer, SortOption> getSortOptionsMap()
10         throws SortException{
11       // Operations using sortOptionsMap class variable
12       synchronized (SortOption.class){
13          //Synchronized operations
14       }
15       return SortOption.sortOptionsMap;
16   }
17   public static Set<SortOption> getSortOptions()
18         throws SortException{
19          // Operations using sortOptionsSet class variable
20          synchronized (SortOption.class){
21             //Synchronized operations
22          }
23          return SortOption.sortOptionsSet;
24       }
25 }
```

Figure 3.25: Self dependencies of `SortOption`.

As mentioned earlier self dependency is concentrated at one point in this case and this self dependency used in different points of software. Transforming this self dependency to a parallelizable construct can be fruitful.

`SortOption` class from DSpace in Figure 3.25 has an RCD value of 0.33 and an NCD value of 2, acts as a mediator between many different sorting implementations in the software.

It is not surprising to see some of its methods having synchronized sections as an outcome of this situation. Although it has a few sections to break self-dependent behavior(NCD), class has a lot more dependencies than self dependency(RCD) making those self dependencies possibly scattered through the software which is relatively bad for parallelization.

3.3.3.3 Bridge metric examples

Utilization strategy of bridge metrics in parallelization can be listed as follows:

1. Most important metric for a bridge is RBSD where a higher value indicates some type of alternating usage most of the time in practice. However this metric should be paid equal attention with RSBC metric. Although having RSBC as low as possible

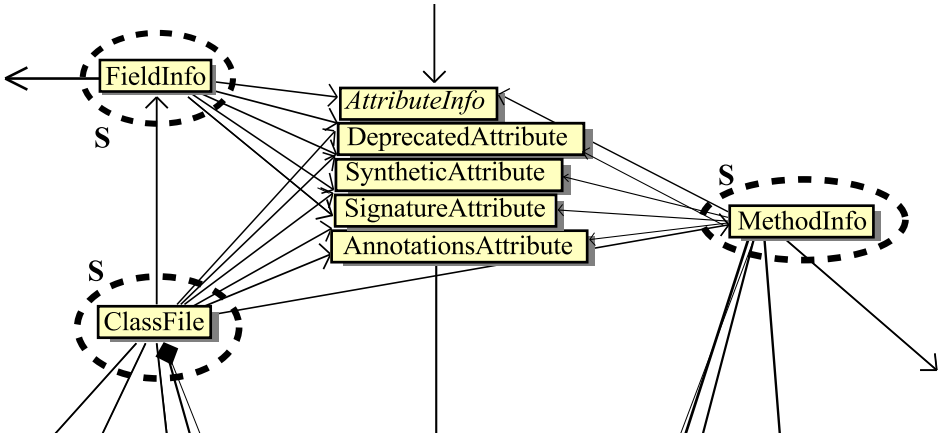
together with a high RBSD is the most favorable case, having a high RSBC may sometimes mislead developer, especially when bridge objects are created using a factory.

2. As mentioned, RSBC is an important metric since bridges mostly show their alternating behavior in a polymorphic way. RSBC should be considered together with number of classes inside the bridge and RBSD value. If bridge both has a high RBSD value and large number of classes one should remember RSBC can be sometimes misleading since polymorphism is not the only way for a software to implement alternating behavior.
3. Lastly REIBD should be considered to fine tune the parallelization of the bridge. While having an REIBD close to -1 is better if the bridge has a low RBSD value since this situation may end up with the isolation of the bridge classes(although in practice this is a rare situation). When the RBSD value is high having an REIBD value closer to zero is better since it indicates fewer number of non-source dependencies most of the time.

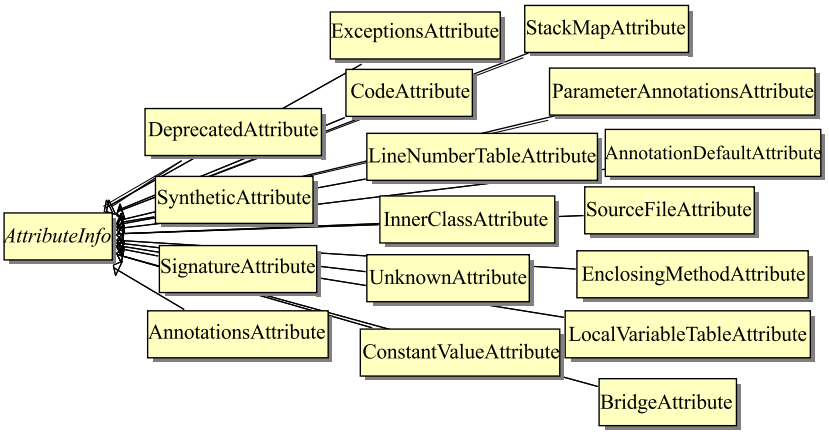
Based on the guidelines enlisted above, example interpretations of bridge metrics from case studies can be given as follows.

An instance of a bridge pattern can be seen in Figure 3.26, where a subset of the descendents of `Attribute` form an authority bridge. At the ends of the bridge, bridge objects are used in a similar way by being alternatingly switched inside loops. Inside `ClassFile`, `MethodInfo` and `FieldInfo` classes, `AttributeInfo`'s subclass instances are kept in a buffer array which are then iterated over by the mentioned loops. This situation needs special care while scattering and gathering the buffer of the bridge objects.

An example snippet of bridge object access can be found in `ClassFile` constructor in Figure 3.27. This method contains a switch statement in which appropriate actions are taken depending on attribute type. This switch is executed many times in a loop for each `AttributeInfo` object. By parallelizing this loop the performance speedup can be seen in Figure 3.16 as the workload of the switching operation increases. This group is actually two overlapping bridges among three classes called `ClassFile` `MethodInfo` and `FieldInfo`. Dominant external dependencies implicated



(a)`AttributeInfo` Dependencies.



(b)`AttributeInfo` inheritance relationships.

Figure 3.26: `AttributeInfo` descendents as an authority bridge instance.

by REIBD(1) metric are dependencies inside another bridge. Having high RBSD(0.63) and low RSBC(0.2) values, this bridge is a good candidate for parallelization. On the other hand, RBSD value for the group is not as high as it should be, a false negative caused by the overlapping bridge connections it have.

```
1 ClassFile::ClassFile(const char* buf, unsigned buf_size)
2 {
3         /*Some processing*/
4         switch (attr -> Tag())
5         {
6                 case AttributeInfo::ATTRIBUTE_Synthetic:
7                     /*Operations using SyntheticAttribute object*/
8                 case AttributeInfo::ATTRIBUTE_Deprecated:
9                     /*Operations using DeprecatedAttribute object*/
10                case AttributeInfo::ATTRIBUTE_Signature:
11                    /*Operations using SignatureAttribute object*/
12                case AttributeInfo::ATTRIBUTE_SourceFile:
13                    /*Operations using AnnotationsAttribute object*/
14                /*Several other cases*/
15        }
16 };
```

Figure 3.27: `AttributeInfo` usage.

Another bridge Example from JBoss can be seen in Figure 3.28. In this bridge it can be seen that the bridge has a balanced amount of external/internal dependencies. These non-source dependencies originate from a single class called `JDBCTypeComplexProperty` which is actually used by `JDBCTypeComplex` in practice. If `JDBCTypeComplexProperty` is taken out of the bridge, its metric values are improved at a great amount. Moreover `JDBCTypeComplex` and `JDBCTypeSimple` are sibling classes that are created by a factory called `JDBCTypeFactory` which is a good indication of a bridge usage in practice.

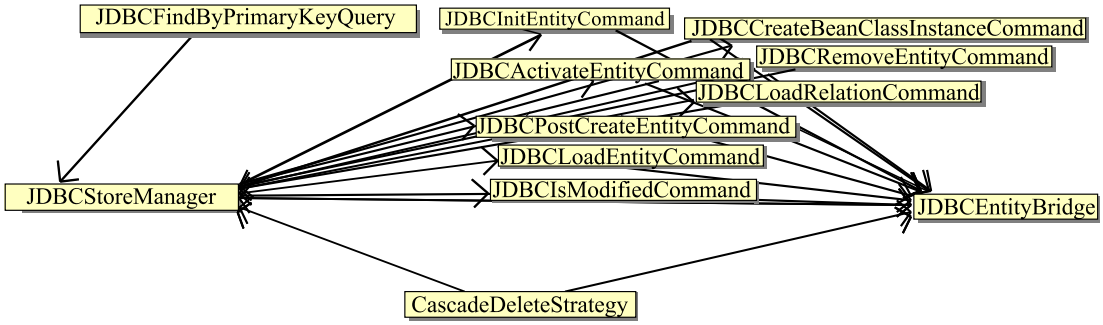


Figure 3.28: An example bridge from JBoss.

A final example bridge from JBoss is in Figure 3.29 which is actually a false negative example for RSBC. In this example, the importance of analyzing the metrics collaboratively can be seen. Even though none of the classes are siblings in this bridge it has a high RBSD value and large number of classes inside the bridge showing a good parallelization opportunity. When the code is analyzed it is no surprise two sources of the bridge use the bridge classes heavily inside them. First of all, objects of the bridge classes are created by a factory inside `startStoreManager()` method of `JDBCStoreManager` class. This can be a good place to introduce parallelism as discussed earlier. For the other end of the bridge, `JDBCFieldBridge` class is used heavily inside the loops of bridge classes as well.

3.3.3.4 Island metric examples

REEID and CIID can be analyzed together to detect the islands that are connected to the rest of the software over local authority/hub classes(like `JDBCEntityMetaData`) to draw guidelines on parallelizing modular parts of software. A lower REEID is more important in any case where the island is more independent.

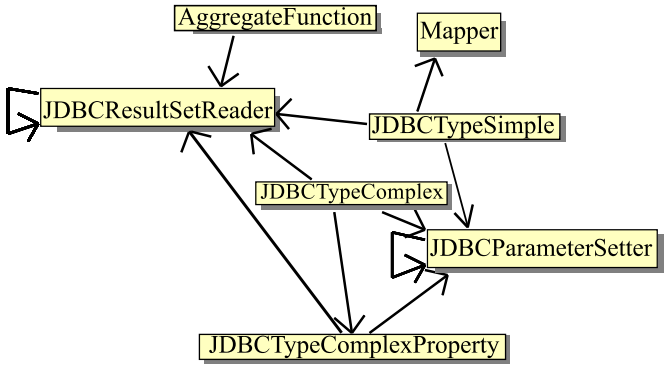


Figure 3.29: An example bridge from JBoss.

Having different CIID values can have different advantages: an island with a low CIID can be packed easier but harder to parallelize especially if it has a large number of external dependencies. On the contrary, additional local parallelization strategies can be applied on specific classes inside the island if CIID is higher.

Island metrics involve large number of classes and mostly self-defining most of the times. For instance, REEID metric is a natural outcome of diagram clustering; it can be easily inferred that having large number of external dependencies making the group more dependent to the rest of the software. On the other hand it may be useful to look at two examples of CIID metrics in the case studies.

In Figure 3.30, first island has a CIID value of 3.36 indicating the heavy dependency load on two classes called `JDBCEntityMetaData` and `ApplicationMetaData`. On the other hand, for the second island this value is 1.3 indicating a well balanced dependency distribution which can be seen in the figure as well. On the contrary REEID metric of the second island is about three times higher compared to the first island making it more dependent to the rest of the software. Similar conclusions can be made, visually analyzing the figure, where almost all of the external dependencies that the first island has is owned by `JDBCEntityMetaData`.

3.4 Detecting Dependency Patterns

By defining dependency patterns and their properties it is possible to perform a structural parallelization operation over sequential software. On the other hand detecting those patterns inside class diagrams may not always be performed easily all the time. Especially for the specific pattern ”bridges“ conventional techniques provide

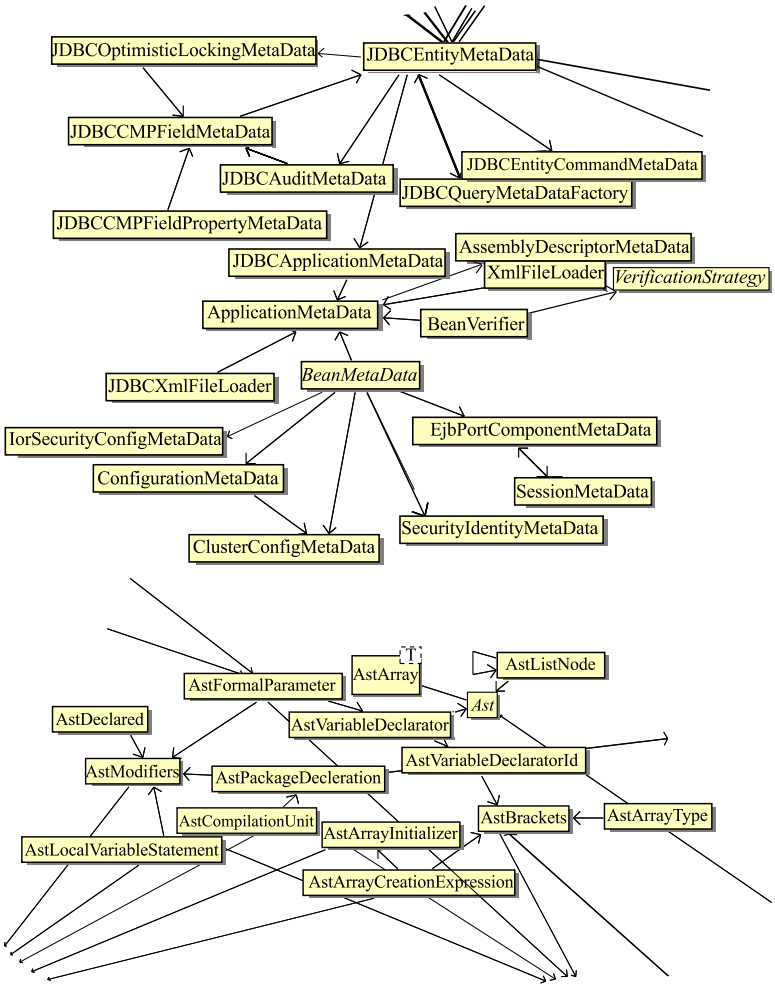


Figure 3.30: Sample islands having distinct island metric values.

an inadequate performance. In this section an enhancement over clustering techniques is presented to discover the dependency patterns inside class diagrams.

3.4.1 Related work on pattern detection

Graph clustering has been applied previously to software models for modularization aspects and static analysis of software. [82] used hierarchical graph clustering over dependency graphs of software files in order to reorganize the modular structure of software: In a graph constructed from software modules, the connectivity of vertices inside/among clusters is used in optimizing the modularization of software. [83] used spectral graph partitioning techniques in order to detect reusable components in software by analyzing class diagrams. This approach is based on an iterative method for partitioning class diagram in order to identify dense communities of classes.

By conducting a more specific analysis on dependency graphs extracted from software, it is possible to reason about many different aspects of object oriented systems including software quality, modularization, and runtime properties. [84] apply clustering to dependency graphs extracted from Java source code to increase modularity. [85] performs dependency analysis at the module level in order to reveal the high level structure of software. A structural visualization was accomplished by partitioning the graphs constructed from module-level inter-relationships obtained from source code analysis. [86] used dynamic dependencies to construct a more realistic dependency graph from pure static representations of the software as input to clustering. This approach can be used for program comprehension, but it cannot be applied during early stages of software development since source code and/or dynamic information is required. In contrast, [49] built a weighted communication graph using predetermined rules at the design stage. This graph was than partitioned in order to minimize the communication cost among clusters.

Software design models can also serve as the source of graphs enabling us to reason about design-level aspects of software. UML class diagrams are one of the most widely used tools to model the static structure of software. As there are many different relationships among classes inherent in UML class diagrams (such as composition, generalization, or association) various mappings of the diagram to a graph can be performed to extract dependency graphs through graph clustering. Using this approach, Wu analyzed UML class diagrams to support program slicing and coupling measurement [87]. Similarly, [58] presented graph theoretical techniques as a generic way to discover patterns in UML diagrams, albeit considering any relationship between two classes as an edge in the graph.

In most of this work, clustering has been applied to dependency graphs without considering structures that emerge from software design. For example, albeit [58] identify highly coupled, huge classes to which they refer as “god classes”, their work does not comment on utilizing these structures during graph clustering. In contrast, in this section common dependency patterns that emerge in UML class diagrams are focused on improving the performance of popular clustering techniques when detecting those patterns.

3.4.2 An enhancement to graph clustering for dependency pattern detection

Graph theory and clustering have been applied to many different aspects of software analysis. In particular, dependency graphs are widely used in the analysis of object oriented software systems, treating software artifacts as vertices and relationships among them as edges. The dependency graph is extracted from a program using various methods, including source code and byte code analysis.

UML [88] has become the most prevalent visual modeling language for software development. As such it is also the platform of choice for performing analysis of object oriented software designs. In particular, class diagrams have been the subject of clustering techniques.

Clustering studies applied to software designs usually deal with static properties of software like modularization [84] and software structure [85]. To reason about dynamic properties of software, the analysis should include runtime information which is not present at the early design stage. By detecting recurring class diagram structures (which is referred to as dependency patterns) and their runtime properties it will be possible to relate them to dynamic properties of software without having the actual implementation and/or runtime information.

However, certain structures in class diagrams are frequently missed in the clustering process because they do not fit neatly into a the definition of a cluster. Typically these structures are comprised of a group of classes having dense identical dependencies towards or from two specific classes outside the group. Class groups with such dependencies are referred as *bridge* patterns in this thesis. Current clustering techniques tend to merge bridge patterns with larger class groups or distribute the classes inside the bridge pattern amongst many other class groups.

In related research on parallelization of software designs it has been found that bridge patterns play a key role, and therefore, methods of detecting such patterns are investigated.

In thesis studies class diagrams are represented with undirected graphs considering only dependency relationships among classes. Clustering methods are extended with algorithms that are able to cope with patterns that were not able to be detected as clusters, independent of the particular clustering method being used. The proposed

algorithm is focused on detecting bridge patterns. Without this step, key aspects of the relationship between elements in a UML class diagram will be missed in an analysis of these diagrams, as the bridge pattern does not fit the definition of a cluster.

3.4.2.1 Clustering for dependency patterns

In this section, clustering is leveraged to identify dependency patterns and apply different graph clustering techniques to graphs extracted from class diagrams. Although dependency is a directed relation, detection of dependency patterns is implemented using undirected graphs, due to its superior clustering performance. Using undirected graphs does not interfere with pattern direction detection; directional analysis of patterns can still be performed independently after pattern detection.

This approach is evaluated by searching for dependency patterns inside the open source compiler project Jikes [57] (which originated from the IBM alphaWorks project). Class diagrams for Jikes' `CLASS` (39 classes), `LOOKUP`(41 classes), and `AST` (103 classes) packages are obtained by reverse engineering from header files, resulting in medium to large size diagrams. In Appendix B graphs extracted from dependency diagrams of `LOOKUP` and `AST` can be seen. In these graphs, many occurrences of dependency patterns can be spotted easily. Some patterns are labeled in these figures and will be referred to in the discussion of the experiments in later sections.

In order to detect these patterns automatically, the following graph clustering techniques are applied to undirected graphs extracted from the dependency diagrams: (i) k-way hierarchical graph clustering [59], (ii) clustering based on computing normalized cut and ratio associations for a given undirected graph without eigenvector computation [89], (iii) spectral graph clustering [60], and (iv) Markov clustering and flow simulation [61]. Clustering experiments were conducted using the software tools Cluto [90], Graclus [91], kernlab [92], and MCL [93], respectively.

In Figure 3.31, the results of automated clustering applied to `CLASS` using spectral graph clustering (a) and Markov clustering (c), respectively, are compared to a manual clustering of the same graph (b).

Table 3.2 shows the adjusted rand index [94] for these experiments which provides a basic comparison between the results of the various clustering techniques and the desired clusters. The adjusted rand index is a measure of the similarity between two

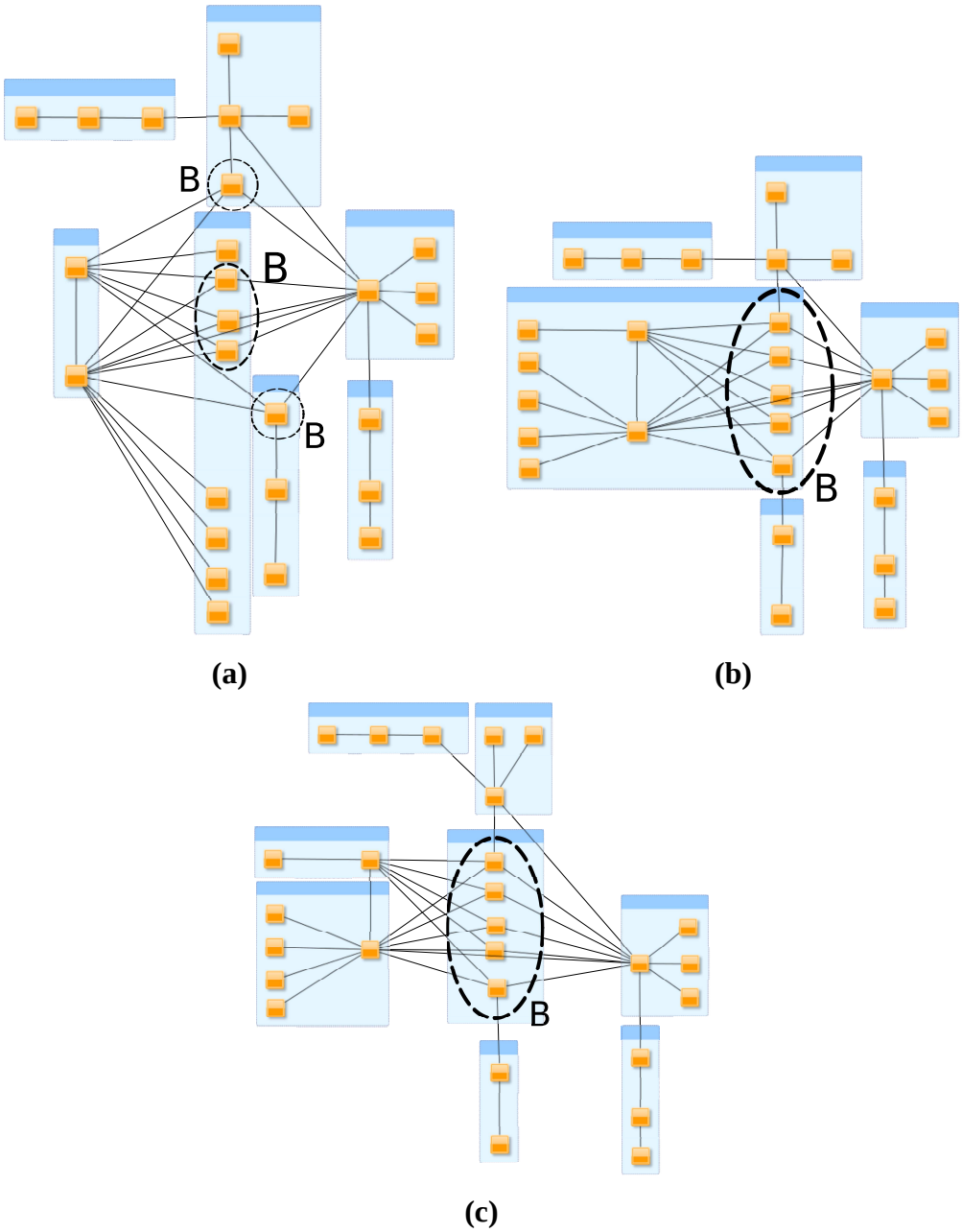


Figure 3.31: Performance of spectral graph clustering (a) and Markov clustering (b) compared to manual clustering (c).

data clusterings, yielding a value between 0 and 1, with 0 indicating that the two data clusters do not agree on any pair of points and 1 indicating that the data clusters are exactly the same.

As the results shown in Figure 3.31 reveal, the studied clustering techniques are not very successful in partitioning the dependency graph of the case study. Examining the obtained clusters in detail, one can see that these clustering techniques were not able to detect any bridge dependency patterns. For example, in the manually created target

clustering (b) a bridge pattern can be seen marked as group B. In (a), it can be see that two vertices of the bridge are scattered amongst other clusters and in (c) this bridge is merged altogether with another cluster.

Table 3.2: Adjusted rand index metric obtained for the studied clustering techniques: Hierarchical graph clustering, clustering with normalized cut and ratio associations, spectral graph clustering, and Markov clustering.

	CLUTO	GRACLUS	KERNLAB	MCL
CLASS	0.182	0.474	0.516	0.558
LOOKUP	0.353	0.343	0.262	0.183
AST	0.481	0.333	0.540	0.150

The reason for the failure to detect bridges is the loose relationship of vertices within the bridge pattern as well as their defining connections to single vertices outside of the cluster. Therefore it is needed to provide a detection technique that is able to separate bridge classes from other clusters. For the rest of the dependency patterns described in Section 3.2, clustering techniques provide acceptable performance.

3.4.2.2 Bridge detection algorithm

An algorithm is presented to find bridge patterns in a dependency graph derived from class diagrams. As defined, bridges are groups of classes where all classes inside the group are connected to at least two common classes. In a class diagram, there may be overlapping bridges where two bridges share a class or a group of classes. Classes inside a bridge may have dependencies between each other or with other classes outside the group. In practice bridges have no or few dependencies other than those to the classes they connect.

The proposed algorithm uses the Hamming distances among the vertices in the adjacency matrix of the undirected and unweighted graph extracted from dependency diagram. The Hamming distance between two strings of equal length measures the minimum number of substitutions required to change one into the other. The algorithm also accepts a threshold parameter which determines the required dependency similarity of vertices inside a bridge. It is assumed that authority and hub vertices have been excluded as well as those vertices that have only a single connection. Detection of authority-hub vertices and singly connected vertices can be performed simply by counting the number of edges originating from each vertex.

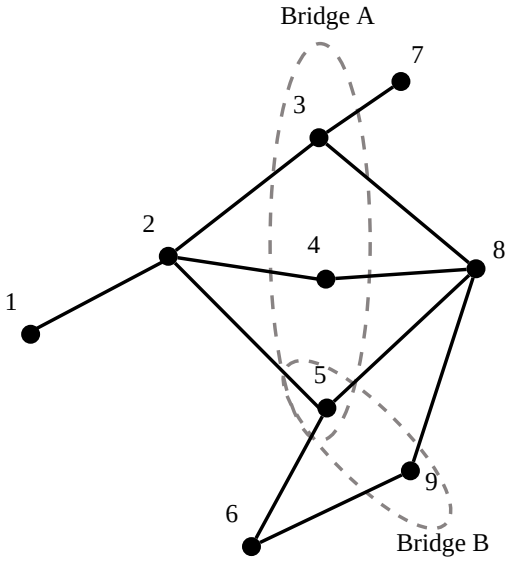


Figure 3.32: A sample graph to be used in illustrating bridge detection algorithm.

The proposed algorithm shall be illustrated using the sample graph shown in Figure 3.32. In the sample graph, a bridge can be seen comprised of the vertices labeled 3, 4, and 5(Bridge A), connecting vertices 2 and 8 and a bridge consisting of vertices 5 and 9(Bridge B), connecting vertices 6 and 8. These two bridges overlap as they share vertex 5. The vertices labeled 2 and 8 are excluded as candidates for a bridge since they are determined to be authorities or hubs; the vertices numbered 1 and 7 are excluded since they have single connections.

In Algorithm 1, the distance matrix is formed by calculating Hamming distances between each vertex in the adjacency matrix of the graph in Figure 3.33(a). The distance matrix, shown in Figure 3.33(b) for the sample graph, is obtained in lines 2 through 5 of Algorithm 1. Those pairs of vertices that are similar to each other below a given threshold are detected in the next step(line 6). The Hamming distance is used as the similarity measure between vertices since it counts the number of different connections between two vertex rows in the adjacency matrix.

In the example, a threshold of 2 is used, which means the only distances selected are smaller than or equal to 2. The threshold value used in our algorithm indicates the maximum distance between two vertices within a bridge. In our experiments, setting this threshold around 2 yielded best results. The threshold may need to be adjusted for different scenarios, based on experiments.

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	1	0	1	1	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0
4	0	1	0	0	0	0	0	1	0
5	0	1	0	0	0	1	0	1	0
6	0	0	0	0	1	0	0	0	1
7	0	0	1	0	0	0	0	0	0
8	0	0	1	1	1	0	0	0	1
9	0	0	0	0	0	1	0	1	0

(a) Adjacency matrix.

	1	2	3	4	5	6	7	8	9
1	0	5	2	1	2	3	2	5	3
2		0	7	6	7	4	3	2	6
3			0	**1**	**2**	**5**	4	7	**3**
4				0	**1**	**4**	3	6	**2**
5					0	**5**	4	7	**1**
6						0	3	2	**4**
7							0	3	3
8								0	6
9									0

(b) Hamming distance matrix.

	1	2	3	4	5	6	7	8	9
$3-4(BridgeA)$	0	1	0	0	0	0	0	1	0
$3-5(BridgeA)$	0	1	0	0	0	0	0	1	0
$4-5(BridgeA)$	0	1	0	0	0	0	0	1	0
$5-9(BridgeB)$	0	0	0	0	0	1	0	1	0

(c) Common connections of similar vertices.

Figure 3.33: Matrices of the sample graph in Figure 3.32 used in bridge detection.

Algorithm 1 Bridge discovery using the adjacency hamming distance between vertices.

1: Form the adjacency matrix of graph $G(v,e)$
2: **for all** pairs of vertices **do**
3: Find Hamming distances between corresponding rows
4: **end for**
5: Form distance matrix between vertices
6: Detect pairs below a given threshold and add them to set $P(v_1,v_2)$
7: **for all** pairs (v_i,v_j) in set P **do**
8: Add (v_i,v_j) and $\bigwedge$(row(v_i),row(v_j)) to P'
9: Remove pairs from set P' which have only one common connection
10: **end for**
11: **for all** joinable pairs $(s_1 \ldots s_n)$ in set P' **do**
12: Add $\bigcup(s_1,..,s_n)$ and $\bigwedge$(row(s_1),...,row(s_n)) to result set R
13: **end for**

As mentioned earlier, vertices numbered 1, 2, 7 and 8 are excluded from this operation as well as the diagonal 0's which indicate self connectivity between vertices. Distances to which the selection process is applied to are shown in bold in Figure 3.33(b). After the selection process, the following vertex pairs are selected: $3-4$(in Bridge A), $3-5$(in Bridge A), $4-5$(in Bridge A), and $5-9$(in Bridge B). The pairs that reside in the same bridge are then merged into a common node set of the bridge.

In the loop starting at line 7 of Algorithm 1, the common connection matrix of selected vertex pairs is constructed, see Figure 3.33(c). The matrix is built up using binary ANDed rows of pairs from the adjacency matrix indicated by $\bigwedge$). In this matrix, each pair that has two or more common connections is joined until no further vertices can be joined. Pairs with only one common connection outside the pair are eliminated (line 9), since vertices inside a bridge need to have two common connections outside the bridge.

When the loop starting at line 11 of Algorithm 1 is reached, each line in Figure 3.33(c) is joined with suitable rows inside the matrix. The join operation is performed over two common connections in rows being joined. For example, the first three rows of Figure 3.33(c) can be joined over columns 2 and 8 while the last row $(5-9)$ cannot be joined with any other row since there are no rows which have connections in the sixth and eighth columns. After the join operations, groups $3-4-5$ and $5-9$ are detected as bridges. The join operation over rows is performed by applying set union (indicated as $\bigcup$) over vertex sets and applying binary AND to corresponding rows. An additional threshold could be applied to eliminate bridges below a desired size.

This algorithm has time complexity polynomial in the number of all vertices v inside bridges. The joining operation starting at line 11 in Algorithm 1 operates on v^2 vertex pairs resulting in a complexity of $O(v^4)$. Such complexity would not be acceptable for large data sets. However, it is not common for a class diagram to have thousands of classes, and there are significantly fewer classes involved in bridges than the total number of classes in a diagram. Consequentially, this algorithm is usable in practice.

3.4.2.3 Evaluation of bridge detection algorithm

The graph obtained by applying bridge detection algorithm to the case study is shown in Figure 3.34. Figure 3.34(b) repeats the target clusters for `CLASS` as determined by

manual clustering, with the bridge highlighted and labeled as B. Figure 3.34(a) shows the result of clustering with MCL, which yielded the best result among the studied clustering techniques for `CLASS`. In (a), the bridge B is merged with two other clusters. If the number of clusters are increased, the vertices of the bridge would once again be split over separate clusters as shown in Figure 3.31(a). Figure 3.34(c) shows clusters resulting after detection of authority or hub vertices and detection of bridges using bridge detection algorithm on the clustering obtained from MCL.

For CLASS and LOOKUP, bridge detection algorithm was able to find the bridges that were detected manually. In the more complicated AST graph, bridge detection algorithm in addition found some smaller bridges (consisting of two classes) that were spread throughout the diagram. The threshold value used in bridge detection algorithm indicates the maximum distance between two vertices within a bridge. In the experiments, setting this threshold around 2 yielded best results. The threshold may need to be adjusted for different scenarios, based on experiments.

Table 3.3 shows the adjusted rand index when applying bridge detection algorithm to clustering obtained from the studied techniques. Except for AST, the algorithm nearly doubled the adjusted rand index scores. To put the minimal improvement for AST in context, consider that the adjusted rand index calculation used in this evaluation does not take overlapping bridges into account. A visual inspection of the results for AST, however, indicates significant subjective improvement, see Figures B.4 B.5 B.6.

Table 3.3: Adjusted rand index for clustering improved by applying the bridge detection algorithm to studied clustering techniques.

	CLUTO	GRACLUS	KERNLAB	MCL
CLASS	0.925	0.925	1.000	1.000
LOOKUP	0.699	0.692	0.654	0.545
AST	0.511	0.549	0.544	0.460

The improvement over clustering results by the bridge detection algorithm is somewhat expected, since bridges do not fit the definition of a cluster. Vertices inside a bridge are typically less connected amongst each other and their connections outside the group are all to specific vertices.

There are some trade-offs to be considered when deciding on thresholds for bridge and authority or hub detection. One may end up with a larger set of bridges including

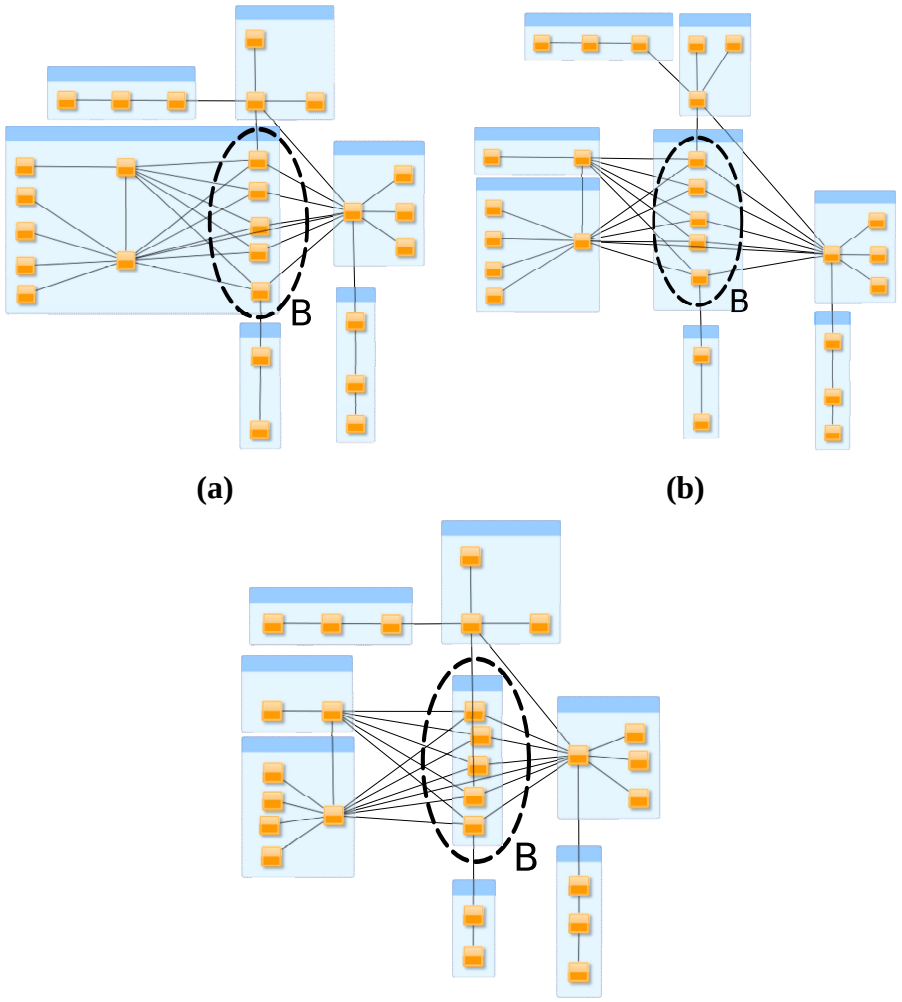


Figure 3.34: Clustering obtained from MCL (a), manually (b), after detecting and separating bridges and hubs/authorities from MCL results (c).

practically useless ones when thresholds are chosen loosely. On the other hand, it is possible to miss some of the bridges if thresholds are set too tight.

Appendix B compares manually obtained target clustering, the best obtained clustering, and the result of applying bridge detection algorithm to the best obtained clustering for `LOOKUP`. Clustering in Figure B.2 is the output produced by Cluto, which yielded the best result for `LOOKUP` as shown in Table 3.2. It is clearly seen that in this clustering, the bridge indicated as group F is merged with the vertices it is connected to, while the bridge indicated as group B in Figure B.1 is split. After applying bridge detection algorithm to Figure B.2, as shown in Figure B.3, the two bridges are isolated as B and F. Only one vertex from F remains separated, yielding a lower error rate in bridges as compared to the clustering in Figure B.2.

Appendix B contains similar results for AST. The clustering produced by kernlab in Figure B.5 separated one vertex from group B as compared to the desired clustering Figure B.4. Moreover, the clusters labeled D and E are merged completely with other clusters. After applying the bridge detection algorithm to Figure B.5, group D and group E are separated into different clusters in Figure B.6. Also, group B is split into three different bridges where group B1 and group B2 share two vertices inside the bridge and group B1 and group B3 share one vertex inside the bridge. This is an example of overlapping bridges which, of course, cannot be detected by current clustering since by definition clusters may not overlap.

3.5 Summary and Conclusions

It is possible to achieve performance boosts by ignoring parallelism inherent in the problem and instead applying local imperative parallelization techniques. Another source of parallelism, however, is implicit in the software structure, which can be revealed through dependencies between classes represented in static software models. Focusing on the these models has the potential to detect parallelism at the design stage which is harder to detect using code analysis.

In this chapter, static class diagrams are analyzed to determine portions of the diagram that will exhibit distinctive properties at runtime that make it amenable to parallel execution. Dependency patterns are defined and identified their usage in class diagrams showing through experiments that these patterns play an important role in parallelization of object oriented software. As illustration, examples are presented from design patterns and a case study, which exhibit these dependency patterns and demonstrate how these patterns lead to select particular techniques for parallelizing the represented software. Detecting instances of dependency pattern inside software design patterns is an indication that dependency patterns occur in object oriented software frequently.

By analyzing the static structure of software models it is possible to detect opportunities for parallelization and to provide guidelines for injecting parallelism into the software under development. These opportunities are represented by patterns in the design model such as “sibling bridges”, “master classes”, “one authority to many subclasses”, etc. These patterns are not only useful in injecting parallelism but also

point to areas of the software model that need to be synchronized, such as “authority superclasses” or “one authority to many subclasses”, or to areas that have sequential behavior (“self-dependent classes”). Software performance being heavily influenced by its implementation, using dependency patterns provides guidelines on how to benefit from software designs to lead the programmer in implementing parallelism more effectively.

Later, a set of metric definitions are presented for measuring the properties of dependency patterns. Previous studies on multi-class metrics are bound to package based metrics whereas parallelization metrics are concentrated on performance measurement. In thesis studies the old metrics were adapted and new metrics were derived for dependency patterns. By this it is possible to use dependency patterns as a bridge between software design metrics and parallelizability of software.

Using the metric set proposed, some examples have been presented from four different industrial sized systems, commenting on the validity of the metrics using parallelization experiments and manual code analysis. A correlation study for the metric set is also presented to show that the metrics cover different properties of the dependency patterns.

Finally, dependency patterns are detected using clustering methods over dependency graphs obtained from class diagrams. However, the studied clustering techniques could not identify bridge patterns as these do not fit the definition of a cluster. A bridge detection algorithm was proposed in order to improve the clustering performance for dependency graphs. Applying bridge detection algorithm to clustering results obtained for the case study yielded a noticeable improvement in clustering performance.

Putting all together, in this chapter recurring structures in software design were identified which can be utilized in detecting implicit parallelism in object oriented software. A set of metrics is proposed for more detailed analysis of the dependency patterns and usage of clustering methods in detecting those patterns are elaborated by a bridge detection algorithm. By taking advantage of the dependency patterns proposed in this capter, following chapter aims at performance improvement of object oriented software scheduling.

4. CACHE-AWARE SCHEDULING OF OBJECT ORIENTED SOFTWARE FOR MULTICORE SYSTEMS

In the previous chapter a methodology based on the dependencies among components of an object oriented software is proposed. This methodology can be used to simplify the process of exploring major level parallelization opportunities in a software before the implementation process begins. In this chapter, a step is taken further to investigate the possibility of inferring scheduling information based on the software models. To produce such information, data sharing behavior between classes of the software is examined. Prediction of data sharing behavior of software before runtime lets us to produce a dispatching mechanism based on the software model which utilizes the cache usage and provide performance gains up to 25%.

In Section 4.1 the enhancement philosophy is introduced supported by results obtained by re-scheduling design patterns considering cache-reuse. In Section 4.2 an object dispatcher is presented which uses the cache-reuse policy presented in the first section. Proposed dispatcher uses dependency pattern diagram of the software to match with the processor-memory hierarchy of the processor at hand. Experiments are performed on an image filtering software to reason about the applicability of the cache-aware scheduling.

4.1 Cache-Aware Scheduling of Design Patterns in a Multicore Processor

Improving operating system schedulers to take cache utilization into account is being heavily studied by the community. In most of the studies, a single centralized solution to replace the scheduler is proposed using data gathered from runtime profile of software [28–33] [38–40]. Since proposed improvements are at operating system level, software analysis are carried on lower level software structures like loops or thread groups.

Apart from approaches based on modification of operating system's scheduler, another idea is guiding the scheduler using classes as higher level software components. In

the following sections it is shown that extracting such guidelines from object oriented software design can improve Linux's completely fair scheduler(CFS). Cache-aware scheduling approach is applied on design pattern implementations and performance improvement is gained when the scheduler is guided regarding coupled classes of software. Coupled classes access methods of each other frequently, raising the probability of shared data between their objects at runtime. Design patterns (which can be found frequently in object oriented software) are used to reason about possible object tuples that frequently share data at runtime.

At the end of the experiments it can be seen that extracting information from the software model and placing tightly coupled objects into neighboring cores (cores that share the same cache) improves operating system's scheduler performance. Cache-aware scheduling approach does not need to change the whole scheduling mechanism of the system. Instead it can be applied by analyzing the dependency relation among classes in the class diagram of software and provide a set of candidate cores for the classes that have the potential to communicate frequently at runtime. Placing those classes' objects at neighboring cores decrease cache miss rates by taking advantage of shared data between software classes.

4.1.1 Cache-aware scheduling

In the context of thesis studies, the term *Cache-Aware Scheduling* is used to indicate the operation of guiding operating system's scheduler with the information of shared data between software classes. Shared data can be detected dynamically via runtime environment or an external dynamic analysis tool. However partial or full development of the software at hand is needed to perform this kind of analysis. Software models and static class diagrams are used to reason about parallelism at an early stage of software development.

Using software models to guide scheduling provides two important advantages. Firstly, parallelization information can be obtained before the actual software runs or even before it is implemented. This helps us to design more competing software for multicore systems and to produce parallel code that performs better on different multicore architectures. Secondly, the ability to guide the operating system's scheduler without replacing it during the scheduling process is provided. The analysis of

software model at hand can be performed semi-automatically by a programmer or an automated tool to detect data sharing software components of software. According to this information the operating system's scheduler tries to assign objects, which operate on common data to proper cores so that shared data can be placed into shared caches.

During the analysis of the software three different factors in parallelization should be considered.

- Parallelization : The number of distinct parts in software that can run independently. They should be scheduled to different cores.
- Data sharing : Object tuples that share a significant amount of data regarding shared/non-shared caches. They should be scheduled to neighboring (or same) cores.
- Resource utilization : The ratio of processing cores(not idle) to the number of objects that run on the system. This concept can be seen as the utilization amount of the processing power of CPU at a given time.

Resource utilization is heavily influenced by parallelization and data sharing since these two factors have an orthogonal effect on system performance. Decomposing software too much for the favor of parallelization causes objects to write on different caches frequently and increase cache misses. On the other hand scheduling objects strictly on neighboring cores to utilize cache reuse may cause parallelized objects to wait for the same core even though there are some other idle cores present. This situation decreases the parallelization performance when there exists fewer cores in the die than the objects to be scheduled. During the experiments the effect of each of these factors over another is explored to extract more meaningful information from the model. Practical real-world examples based on design pattern implementations are used, which are small enough to successfully observe the effect of each factor during the scheduling.

In thesis studies Gang of Four (GoF) software design patterns [56] are used to analyze data sharing classes of the pattern. Software design patterns are frequently used in today's object oriented software designs to solve common problems. A large number of studies exist in the literature about detecting software design patterns [95–100], making it possible to automate proposed approach.

Cache-aware scheduling technique is applied on design patterns to show that even for smaller parts of the software a better scheduling can be provided using data sharing information between components. This approach can be applied to larger software where many different instances of many design patterns can be found and analyzed for data sharing. In this chapter it is focused on the applicability of model based scheduler guidance by analyzing data usage of recurring themes in software designs. Cache-aware scheduling approach is not limited to specific software design patterns but rather offers to use parallelization strategies together with patterns that emerge in software designs.

4.1.2 Case studies on software design patterns

Experiments are performed in a system with 4 double cored Intel Xeon processors and an operating system of Linux kernel 2.6 running on it. Java is used as the main programming language to develop the design pattern case studies. Since Java lacks an API to explicitly set a thread's processor affinity, C++ is used to implement *pthread*'s [101] thread affinity setting functions [102] and JNI to call C++ thread affinity setter implementations from Java programs. *pthread* library allows thread distribution via `sched_setaffinity` and `CPUSET` functions which can be used to explicitly define thread-to-processor distribution schemes for the objects in the patterns. For the majority of the experiments, objects of the patterns are programmed as separate threads, and assigned to processors either explicitly under control of the programmer or automatically by the system scheduler.

In the experiments below program runs are repeated for a sufficient number of times to let the running time average converge.

Figure 4.1 presents the central processing unit architecture used in the experiments which consists of four different processors each having two cores with a shared L3 cache of 4096KB in size. In implemented scheduling schemes the term "neighboring cores" is used to indicate the cores that reside in the same physical processor and share the same cache (e.g. core #1 - core #7, core #2 - core #5, core #3 - core #6, core #4 - core #8).

Proposed scheduling approach is abbreviated as CAWS (Cache-Aware Scheduling) where the threads that share data are placed onto neighboring cores as much as

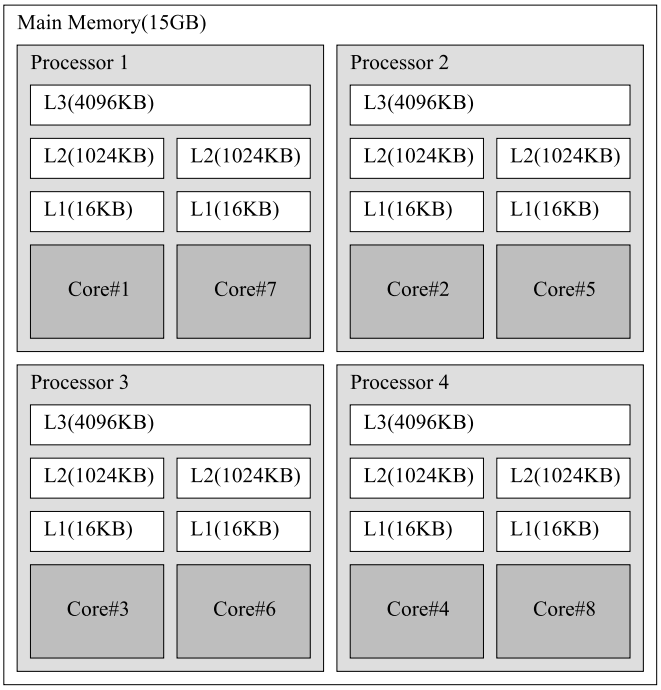


Figure 4.1: Central processing unit architecture used in cache-aware scheduling experiments of design patterns.

possible. Linux's schedulers actually does not take caches into account and migrate the threads often, resulting threads to share caches in a non-determined way.

For the case studies, three different design patterns are implemented: Strategy, Visitor and Observer. All these patterns commonly consist of some master (service requester)-worker (service provider) classes. UML diagrams of the mentioned patterns can be found below.

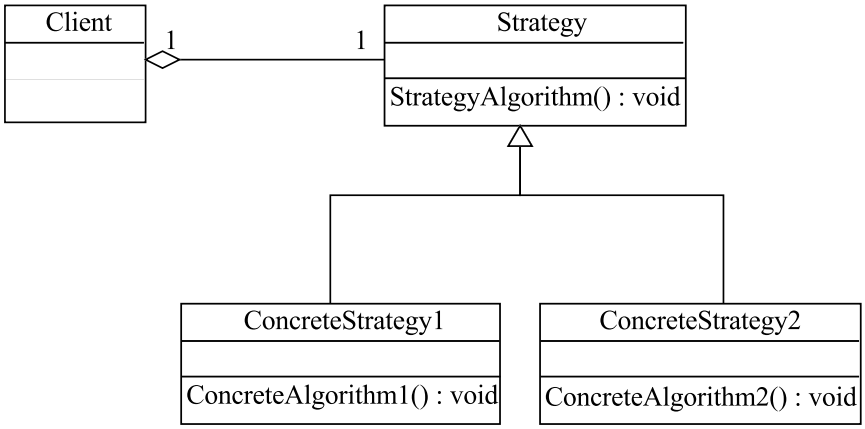


Figure 4.2: Strategy design pattern.

For strategy (Figure 4.2), each strategy object (worker) provides a service of applying a different algorithm on the client (master/service requester) object. Data is shared

between strategy and client objects for this pattern. At runtime there may be many clients (service requester) running in parallel using a specific strategy object in common.

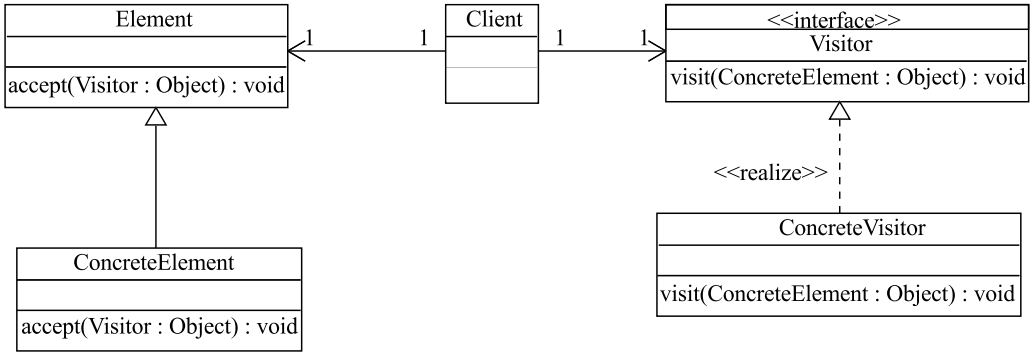


Figure 4.3: Visitor design pattern.

In visitor (Figure 4.3), each visitor object provides its service when it is called explicitly by the master (service requester) object. At runtime there may be many elements requesting services from a set of visitor objects arbitrarily. Some visitor objects may be used in common during these service requests as well. Objects that implement the Visitor interface and Element objects that are visited by Visitors are data sharing components for Visitor pattern.

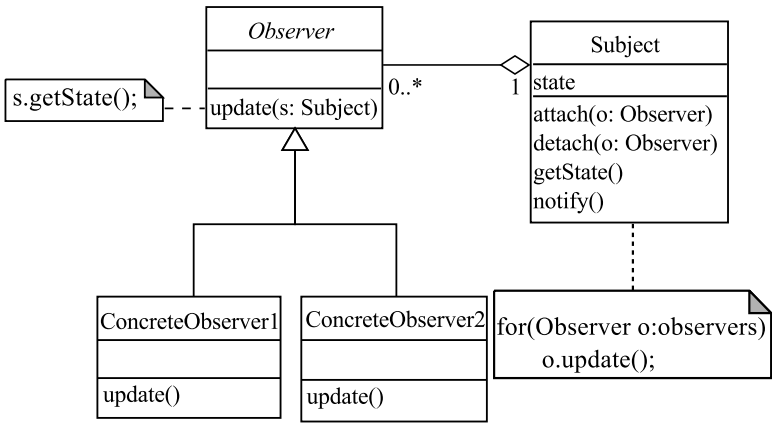


Figure 4.4: Observer design pattern.

In observer (Figure 4.4), a subject object presents the update notification service of its states to a set of observer objects. At runtime some observer objects may register to different common subjects. A Subject object and its observers commonly use the state of the Subject in this design pattern.

Similar examples can be implemented for other patterns as well, the examples in this section are chosen to illustrate different data sharing (read-only, read/write) and

thread creation schemes. Further implementations are explained in detail in Section 4.1.4 but before initiating more complicated experiment scenarios it can be useful to illustrate the effect of cache reuse in scheduling design patterns on basic experimental configuration.

4.1.3 Effects of cache-aware scheduling on basic examples

To show that sharing common caches makes a notable performance difference at runtime a basic set of isolated examples are provided showing the difference of cache-aware scheduling with respect to its counterparts. For this purpose implementations in this section consist of only one master-worker object couple for each design pattern. In each of the examples below there only exist two objects at runtime sharing a fixed amount of data that is proportional to the size of common caches in the processor.

For each set of experiments on a determined amount of data(for each column in tables) the worst-case running times are used to normalize running times between 0 and 1 (worst performance). For each set of experiments CAWS represents cache-aware scheduling policy and CFS is Linux' default scheduler. On the other hand CUS represents cache-unaware scheduling where data sharing objects are always placed at non-neighboring cores. The results obtained for each of the examples are as follows.

- **Strategy :** In Table 4.1 it can be seen that for a large quantity of shared data, scheduling two objects at neighboring cores(CAWS) outperforms the CFS and CUS. When the amount of data being shared gets smaller cache sharing effect loses its significance.

Table 4.1: Normalized running times for basic strategy implementation.

Shared Data:	1MB	8KB	None
CFS	0.95	0.99970	0.99965
CAWS	0.87	0.99965	1.00000
CUS	1.00	1.00000	1.00000

- **Visitor :** In Table 4.2 similar results can be seen in Table 4.1. When the amount of shared data gets closer to shared cache sizes using a cache-aware scheduling starts to perform better.

Table 4.2: Normalized running times for basic visitor implementation.

Shared Data:	1MB	8KB	None
CFS	0.81	0.96	0.9998
CAWS	0.77	0.96	1.0000
CUS	1.00	1.00	1.0000

- **Observer :** Finally in Table 4.3 similar results can be seen except this time an additional scheduling scheme has also been added(referred to as SACS(Same Core Scheduling)). Since one observer and one subject cannot run parallel at all they can be placed at the same core at runtime. When placed at same core, with an amount of data small enough to fit the private cache, the system had a superior performance.

Table 4.3: Normalized running times for basic observer implementation.

Shared Data:	1MB	8KB	None
CFS	0.99	1.00	1.00
CAWS	0.87	1.00	1.00
SACS	0.87	0.29	0.99
CUS	1.00	1.00	1.00

As it can be seen from the running times above, scheduling the data sharing objects in a way that allows them to use the same processor cache outperforms the Linux's CFS. It can also be seen that for the objects that have sequential behavior and use shared data, scheduling them at the very same core provides superior performance since it allows storing shared data at private cache of the core.

From basic examples above it can be seen that migrating shared data among processors and re-fetching large amounts of data inside the memory hierarchy are time consuming operations that degrade software performance. By running experiments on multi-object examples sounder comments about cache-aware scheduling can be made on more realistic cases.

4.1.4 Applying cache-aware scheduling

More complicated configurations on design patterns can be experimented to show the difference between cache-aware scheduling and current scheduler of Linux. In this section many objects inside the design patterns interact during runtime using different parallelization approaches. For all the patterns below, different number of objects are instantiated for each different type of class that the pattern contains. Each object

is implemented as a separate thread, hence two terms (object and thread) are used interchangeably in this Section.

In all the plots presented below y-axis represents normalized runtime performance where normalization is performed by calculating for each experiment.

$$\rho_i = \frac{1}{T_i} \tag{4.1}$$

$$\rho_n = \frac{\rho_i}{\rho_i^{(best)}} \times 100 \tag{4.2}$$

In Equations **(4.1)** and **(4.2)**, T_i represents avarage running time for each case, ρ_i represents performance of each case and $\rho_i^{(best)}$ is the best performance(lowest T_i, highest ρ_i) among all measurements for the plot at hand. Multiplicating the result by 100 enables to easily read the performance differences between measurements with terms of percentage.

4.1.4.1 Strategy

For strategy pattern, a constant number of strategy objects are constructed, each representing a different strategy for a specific number of client objects. Each client object is affiliated with a strategy object at runtime working on a predetermined amount of shared data that is smaller than the size of the shared cache. For the sake of simplicity, the data of the client is always read (never written) by the strategy for this case.

In Figure 4.5, it can be seen that Normalized running times of 32 client objects under different scheduling policies using different number of strategies. When the number of parallelized parts (strategies) are less than number of distinct processing cores in the system, a performance gain is observed which is caused by reduced missing rate during the data access of threads.

If the number of parallelizable parts exceed the number of cores (8 in this case), scheduler starts to preempt threads and change cache content thus the effect of cache-aware scheduling vanishes for number of strategies more than 8. A speedup of nearly 10% compared to CFS, is present when cache aware scheduling is used during the running time of strategy implementation.

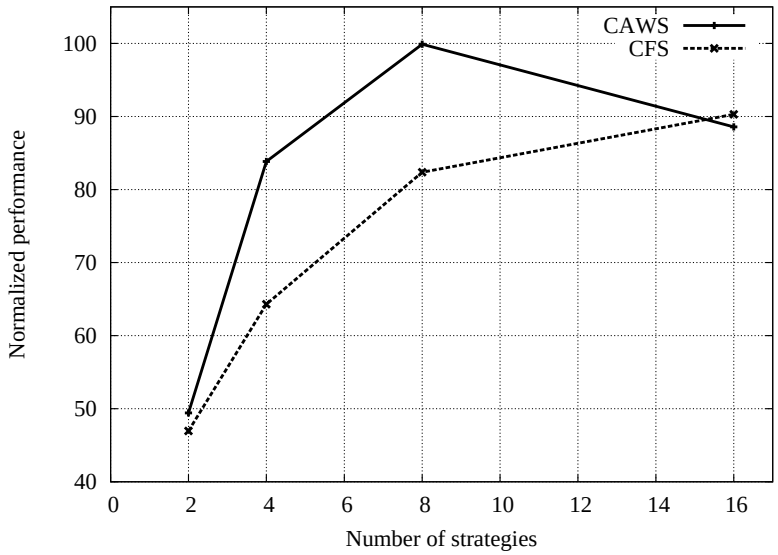


Figure 4.5: Scheduling strategies with different policies.

4.1.4.2 Visitor

In this case, desired number of visitors are constructed independently before element objects run. When an element needs a visitor one is taken from the pool and assigned to the element object. Since waiting times can vary for each element and each visitor, each class holds a queue of the next object to provide/request service. Visitors hold a queue of elements to start serving the next object in line after the ongoing work finishes. A similar situation is present for elements as well, they hold a queue of visitors to ask for a service. For this case a more complicated structure is used where any visitors may visit any elements during runtime; unlike strategy no predetermined element-visitor bindings are applied before system run.

In Figure 4.6 cache-aware scheduling outperformed others until the number of parallelized objects reach the number of cores. Additionally, even when visitors are scheduled on distinct cores from elements but in the same cores with other visitors, CAWS still outperform CFS. This time cache read and writes are used so a cache utilization is not present as much as in strategy case.

4.1.4.3 Observer

Implementation of observer adopts a different object construction approach than the previous cases. This time, observer objects are constructed inside subject objects. This enforces each observer thread to be started and joined inside a different

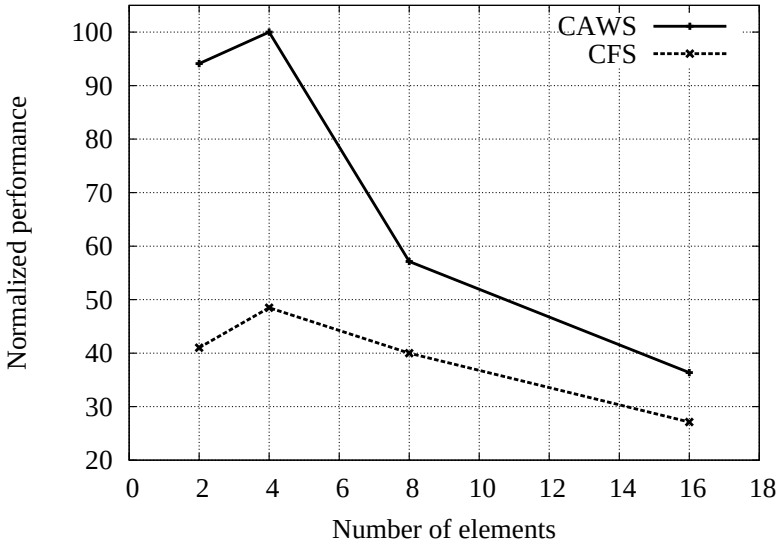


Figure 4.6: Scheduling 8 visitors with different policies.

object, providing larger number of object constructions during runtime. Additionally, subject-observer groups run more isolated in this case thus need less synchronization effort. Moreover instead of enforcing objects to be scheduled on static cores, a set of candidate cores are provided to operating system for each object. Hence a hybrid CAWS-CFS approach is used versus CFS this time.

In Figure 4.7, running times for 2 observers observing different number of subjects is presented. Although observer objects are created and destroyed continuously for each subject, degrading the amount of data reuse during runtime, scheduling the system using a cache-aware policy still provided performance upgrade when compared to CFS.

Finally in Figure 4.8, the number of objects in the system varies as a whole consisting of different number of subjects and observers. Again using CAWS policy results in a better performance than the default CFS scheduling. For both examples mixing CFS with CAWS still provided better results than using only CFS. Albeit gaining relatively smaller performance improvements in some of the cases above, it is important to consider that CAWS operates on application level while CFS operates directly on the kernel level. Guiding operating system scheduler based on model driven analysis may also allow us to start tuning an application for a specific processor architecture before the software is implemented.

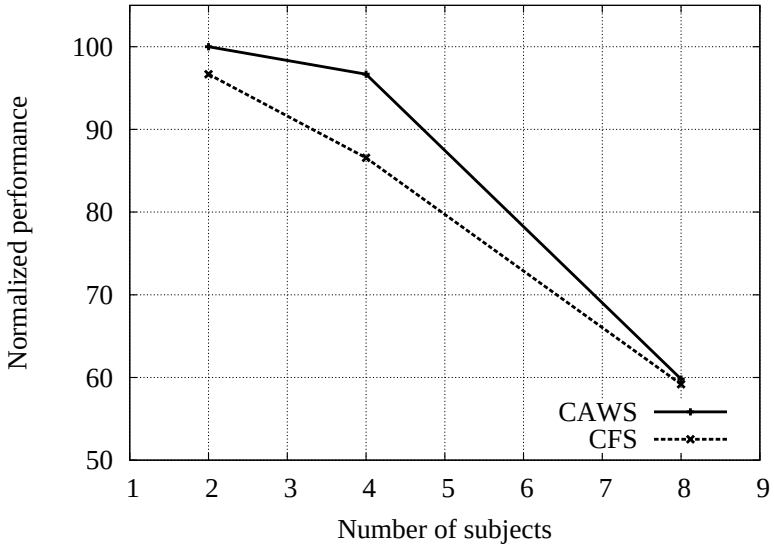


Figure 4.7: Scheduling 2 observers with different policies.

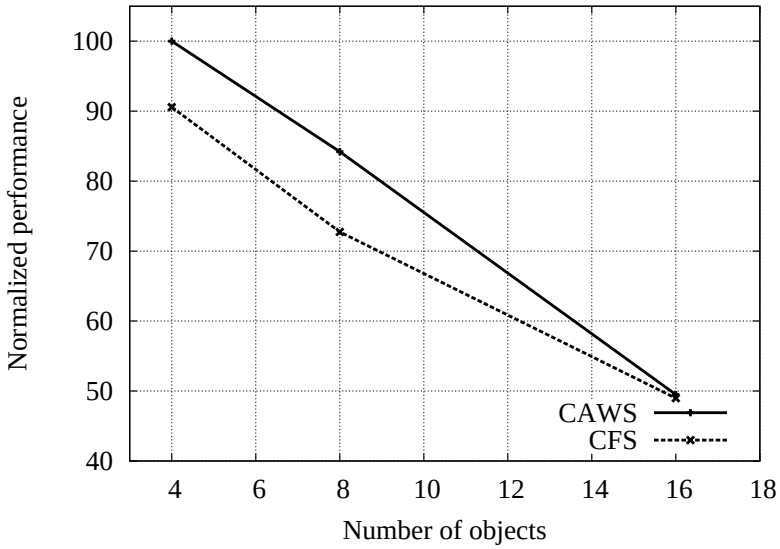


Figure 4.8: Scheduling many subject-observer tuples with different policies.

In experiments with design pattern implementations, the benefit obtained from cache utilization degrades as the number of objects reach beyond the number of cores in the system. This situation is caused by increased number of cache misses as different objects starts to be switched on the cores of the system. Nevertheless this problem loses its significance as the number of cores reside in a chip tend to increase over time.

4.2 A Cache-Aware Dispatcher for Dependency Patterns

Based on the experiments in the last section, scheduling frequently communicating and data sharing objects to the processing elements that share a common cache, provides

performance improvements supporting applicability of CAWS. On the other hand implementing a scheduler from scratch has its own difficulties and problems on quite different domains which are out of the context of this thesis. However even guiding the scheduler using thread affinity directives during runtime provided an acceptable performance improvement for design patterns.

A more systematic and scalable approach for this object dispatching strategy is implemented and experimented in this section. This cache-aware dispatcher uses graph based models of dependency relations among software components and memory-core hierarchy of processing element at hand to provide object/thread-core distribution strategies that will increase cache re-use. Different experiments are conducted for four different software models and two different processor architectures to illustrate the scalability of the proposed dispatcher.

4.2.1 Graph models of dependency patterns and multicore processors

From the initial phases of the thesis studies, software design is presented by labeled graphs and the operations on software like dependency pattern extraction is realized by graph transformations. To provide a scheduling mechanism that maps the dependency pattern onto cores of a chip multi-processor it is enough to perform a graph matching operation between the dependency pattern graph and a processor graph. The processor graph should represent core/memory hierarchy of the chip multi-processor at hand.

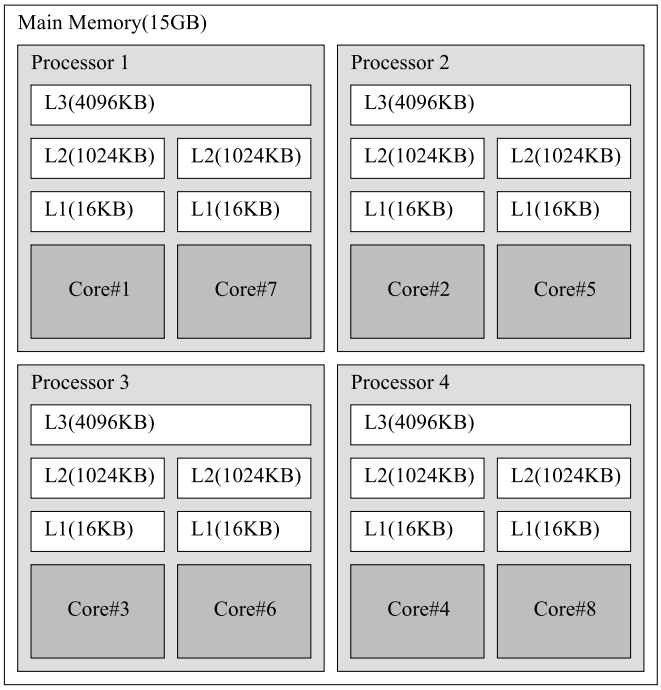
Definition 1. A Processor-Memory Hierarchy Graph(G) is a labeled graph where two types of nodes are called memories(M) and cores(C).

$$G = \langle M \cup C, E \rangle \tag{4.3}$$

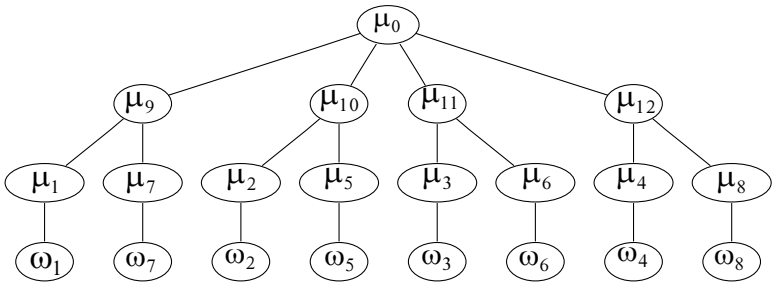
where C $= \{\omega_1, \omega_2 ... \omega_m\}$ is the set of cores in the processors and M $= \{\mu_0, \mu_1 ... \mu_m, ... \mu_n\}$ is the set of memories (private($\mu_1 ... \mu_m$) and shared caches($\mu_{m+1} ... \mu_n$) and main memory(μ_0)). Therefore edges in graph G can be defined as $E \subseteq (M \times C) \cup (C \times M) \cup (M \times M)$

On the basis of this definition an example processor and the memory hierarchy graph representing the processor can be found in Figure 4.9. For the processor in Figure (a) L1 caches are not included in the graph for simplicity.

Graph model of dependency patterns can be defined as follows.



(a) A sample multiple chip multi-processors.



(b) Graph representation of the processing unit in Figure 4.9a.

Figure 4.9: (a) presents an example processing unit and (b) presents the memory hierarchy graph representing it.

Definition 2. A dependency pattern graphs is a labeled graph where five types of patterns called bridges(B), islands(I), authorities(A), hubs(H) and cycles(Y) are represented as nodes in the graph.

$$G = \langle P, E \rangle \tag{4.4}$$

where pattern set(P) is the union of the sets of five different pattern types $P = \{\mathrm{B} \cup \mathrm{I} \cup \mathrm{A} \cup \mathrm{H} \cup \mathrm{Y}\}$. Each pattern set includes a finite number of patterns $m_P \in \mathbb{Z}^+$.

$\mathrm{B} = \{b_1, b_2 ... b_{m_B}\}$

$\mathrm{I} = \{i_1, i_2 ... i_{m_I}\}$

$\mathrm{A} = \{a_1, a_2 ... a_{m_A}\}$

$\mathrm{H} = \{h_1, h_2 ... h_{m_H}\}$

$\mathrm{Y} = \{y_1, y_2 ... y_{m_C}\}$

Each pattern($1 \leq i \leq m_P$) includes a finite number of classes $n \in \mathbb{Z}^+$. Bridges and islands can hold multiple classes and the rest of the patterns can hold a single class.

$b_i = \{{}_bc_1, {}_bc_2 \ldots {}_bc_{n_B}\}$

$i_i = \{{}_ic_1, {}_ic_2 \ldots {}_ic_{n_I}\}$

$a_i = \{{}_ac\}$

$h_i = \{{}_hc\}$

$y_i = \{{}_yc\}$

On the basis of this definition an example dependency pattern graph can be found in Figure 4.10.

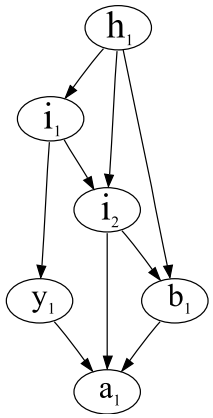


Figure 4.10: An example dependency pattern graph.

4.2.2 A graph matching algorithm for cache-aware dispatcher

Using the graph models of the software and processor at hand, a graph matching algorithm is presented together with a runtime allocation algorithm using the candidate processor groups provided by the graph matching algorithm. Cache-aware scheduler to be presented in the context of the thesis studies firstly applies graph matching algorithm over the graphs at hand to provide a set of candidate core groups. The algorithm that will be used to match the processor and dependency pattern graphs to produce these core groups is presented in Section 4.2.2.1 Those candidate core groups are than used at runtime to place the object at hand to the most idle core present in the set of core group selected for the particular dependency pattern instance.

The most idle core is determined by the runtime resource allocator presented in Section 4.2.2.2. Runtime resource allocator keeps track of all the objects it has dispatched during the program execution and also objects notify resource allocator

when they gave up a core. This enables runtime resource allovator to be informed about idleness of the cores at any given time of program execution. Moreover this kind of information can also be obtained thorugh system calls which is not implemented in the context of this study.

4.2.2.1 Compile-time graph matcher

The graph matcher algorithm is used to provide all of the possible mappings of the dependency patterns at hand to the provided processor architecture. Algorithm 2 starts to place the dependency patterns to the processor graph by calculating candidate core group set of the hub having the most external dependencies initially. Later, dependency graph is started to being traversed using a breadth first search except "node-adjacent" pairs are placed in the "to be visited" queue as "caller-callee" pairs rather than nodes itself. This way, candidate core groups of the callee are set based on the candidate core groups of the caller. This lets the algorithm include different core groups to the candidate set depending on the dependency pattern that accesses the pattern at hand.

At lines 8,14 and 16 of Algorithm 2, a number of sets of candidate cores are determined as the algorithm continues to run. The operations $\uparrow$ and $\downarrow$ are used in the algorithm to select the descendant and ancestor nodes in the memory-core hierarchy. For example, considering Figure 4.9b $\uparrow(\mu_7)$ operation will select the ancestor μ_9 node while $\uparrow(\mu_9)$ will select the set $\{\mu_1, \mu_7\}$. For authorities and cycles sets of candidate cores contains only one core since most of the time they need to be placed in the most idle neighboring core. However, especially for bridges and islands there exist a set of candidate cores where the objects from a specific patterns needs to be distributed at runtime. Following distribution criterion can be used to determine the number of cores that is suitable for the particular number of classes inside a pattern.

$$\delta = \frac{N_c}{N_\omega} \tag{4.5}$$

In Equation **(4.5)**, N_c denotes the number of classes inside the dependency pattern and N_ω denotes the number of cores that share the memory unit represented by a processor's graph node. In order to place a pattern to a group of cores distribution factor δ should be in the interval $[1-\varepsilon_l, 1+\varepsilon_r]$ where ε denotes the threshold that the distribution factor may deviate from the case $\delta = 1$. When $\delta = 1$ each class inside a

Algorithm 2 Graph matching algorithm.

1: $\varsigma = \bigcup(\omega_j)$ a set of candidate cores that a class' object can be placed
2: $\sigma_{P_i} = \bigcup(\varsigma_j)$ is the set of candidate core sets for each pattern
3: Choose the hub($h_{\max}$) with the most external dependencies from dependency graph $G_D(P,E)$
4: $\varsigma_{h_{\max}} = \mu_0$
5: Place each neighbor of ($h_{\max}$) to the visit queue as <c_r,c_e>(Caller class,Callee class) pairs
6: **for all** <c_r,c_e> pairs in the visit queue **do**
7: **if** $c_e \in h_i$ **then**
8: $\mu_{hc_i} = \uparrow(\mu_{c_r})$
9: **else if** $c_e \in a_i$ **then**
10: $\mu_{ac_i} = \mu_{c_r}$
11: **else if** $c_e \in y_i$ **then**
12: $\mu_{yc_i} = \mu_{c_r}$
13: **else if** $c_e \in b_i$ **then**
14: $\mu_{bc_i} = \downarrow(\mu_{c_r})$
15: **else if** $c_e \in i_i$ **then**
16: $\mu_{ic_i} = \downarrow(\mu_{c_r})$
17: **end if**
18: **for all** $\omega_i \in$ C **do**
19: **if** ω_i is a descendant of μ_{c_e} **then**
20: $\varsigma \cup = \omega_i$
21: **end if**
22: **end for**
23: $\sigma_{P_{c_e}} \cup = \varsigma$
24: $\varsigma = \emptyset$
25: Add each neighbor of the c_e to the visit queue as <c_r,c_e> pairs
26: **end for**

pattern is place to a specific core, hence ε_l and ε_r each tune the under-distribution and over-distribution of the pattern classes over the given set of cores.

4.2.2.2 Runtime resource allocator

Runtime resource allocator is used to select the core to schedule each object at runtime. It uses candidate set of core groups provided by the graph matching algorithm in the last section. In order to decide which core to schedule an object to, resource allocator selects a subset of candidate groups by using the calling object's scheduled core information. Among the candidate groups a group is selected based on the core idleness and the object is placed on a core inside the group based on the type of the pattern it belongs to. Resource allocation algorithm is presented in Algorithm 3.

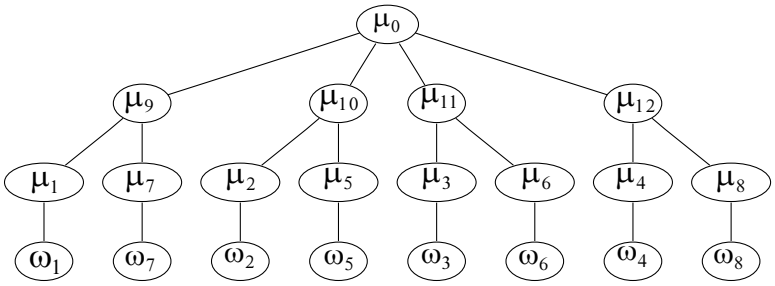
Algorithm 3 Resource allocation algorithm.

1: **while** Software represented by dependency pattern graph $G_D(P,E)$ runs **do**
2: $\varsigma = \bigcup(\omega_j)$ a set of candidate cores that a class' object can be placed
3: $\sigma_{P_i} = \bigcup(\varsigma_j)$ is the set of candidate core sets for each pattern
4: Obtain the<o_r,o_e>(Caller object,Callee object) information of the object to be scheduled.
5: **if** $o_e \in h_i$ **then**
6: **for all** $\varsigma_j \in \sigma_{h_i}$ **do**
7: **if** $\omega_{o_r} \in \varsigma_j$ **then**
8: Place o_e to $\Pi(\varsigma_j)$
9: **end if**
10: **end for**
11: **else if** $o_e \in a_i$ **then**
12: Place o_e to $\omega_k \in \sigma_{a_i}$ where ω_k is a neigbhor of ω_{o_r}
13: **else if** $o_e \in y_i$ **then**
14: Place o_e to most idle $\omega_k \in \sigma_{y_i}$
15: **else if** $o_e \in b_i$ **then**
16: **for all** $\varsigma_j \in \sigma_{b_i}$ **do**
17: **if** $o_r \notin$ H and $o_r \notin$ C and $\omega_{o_r} \in \varsigma_j$ **then**
18: Place o_e to $\Delta(\varsigma_j)$
19: **else**
20: Place o_e to $\Delta(\varsigma_j)$ where ς_j is the most idle set.
21: **end if**
22: **end for**
23: **else if** $o_e \in i_i$ **then**
24: **for all** $\varsigma_j \in \sigma_{i_i}$ **do**
25: **if** $o_r \notin$ H and $o_r \notin$ C and $\omega_{o_r} \in \varsigma_j$ **then**
26: Place o_e to $\Pi(\varsigma_j)$
27: **else**
28: Place o_e to $\Pi(\varsigma_j)$ where ς_j is the most idle set.
29: **end if**
30: **end for**
31: **end if**
32: **end while**

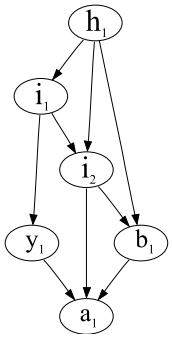
In Algorithm 3 an object may be placed in a specific core, a set of objects may be distributed in a balanced way by CAWS(this operation is indicated as Δ) or a set of objects can be scheduled to a pool of candidate cores(this operation is indicated as Π) where operating system(OS) decides the core that a specific object in the set is going to be assigned to. Also each class in the software marks its affiliated core as busy to the resource allocator before it begins to run and remove the mark after it ends its run. Resource allocator uses these marks to obtain how many objects are set affiliated to a core at a specific time and hence provide a more balanced scheduling scheme.

4.2.2.3 A sample scheduling scenario

It can be useful to illustrate how the scheduler works in a sample scenario. The example processor graph in Figure 4.9 and the example dependency graph in Figure 4.10 will be used in this sample. Both of the graphs can be seen in Figure 4.11.



(a) Graph representation of the processor in Figure 4.9a.



(b) An example dependency pattern graph.

Figure 4.11: Graphs to be used in sample scheduling scenario.

If the graph matching algorithm is run with parameters (ε_l:0.2 ,ε_r:0.5), and the number of classes that i_1,i_2 and b_1 contains is assumed as 6, 2 and 4 consecutively, the algorithm in Algorithm 2 runs in the following sequence.

1. h_1 received
   (a) h_1 is placed to node μ_0
   (b) The group [ω_1,ω_2,ω_3,ω_4,ω_5,ω_6,ω_7,ω_8] is added to candidate set of h_1
   (c) <h_1,b_1> <h_1,i_1> and <h_1,i_2> pairs added to visit queue
2. <h_1,b_1> received
   (a) b_1 can be placed to nodes [μ_9,μ_{10}], [μ_9,μ_{11}], [μ_9,μ_{12}], [μ_{10},μ_{11}], [μ_{10},μ_{12}], [μ_{11},μ_{12}]

(b) Groups $[\omega_1,\omega_7,\omega_2,\omega_5]$, $[\omega_1,\omega_7,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_3,\omega_6,\omega_4,\omega_8]$ are added to candidate set of b_1

(c) $<b_1,a_1>$ pair added to visit queue.

3. $<h_1,i_1>$ received

(a) i_1 can be placed to nodes $[\mu_9,\mu_{10},\mu_{11}]$, $[\mu_9,\mu_{10},\mu_{12}]$, $[\mu_{10},\mu_{11},\mu_{12}]$, $[\mu_9,\mu_{11},\mu_{12}]$

(b) Groups $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6,\omega_4,\omega_8]$, $[\omega_1,\omega_7,\omega_3,\omega_6,\omega_4,\omega_8]$ are added to candidate set of i_1

(c) $<i_1,b_1>$ and $<i_1,a_1>$ pairs added to visit queue

4. $<h_1,i_2>$ received

(a) i_2 can be placed to nodes $[\mu_9]$, $[\mu_{10}]$, $[\mu_{11}]$, $[\mu_{12}]$

(b) Groups $[\omega_1,\omega_7]$, $[\omega_2,\omega_5]$, $[\omega_3,\omega_6]$, $[\omega_4,\omega_8]$ are added to candidate set of i_2

(c) $<i_2,i_1>$ and $<i_2,y_1>$ pairs added to visit queue

5. $<b_1,a_1>$ received

(a) a_1 can be placed to nodes $[\mu_9,\mu_{10}]$, $[\mu_9,\mu_{11}]$, $[\mu_9,\mu_{12}]$, $[\mu_{10},\mu_{11}]$, $[\mu_{10},\mu_{12}]$, $[\mu_{11},\mu_{12}]$

(b) Groups $[\omega_1,\omega_7,\omega_2,\omega_5]$, $[\omega_1,\omega_7,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_3,\omega_6,\omega_4,\omega_8]$ are added to candidate set of a_1

6. $<i_1,b_1>$ received

(a) b_1 can be placed to $[\mu_1,\mu_7,\mu_2,\mu_5]$, $[\mu_1,\mu_7,\mu_3,\mu_6]$, $[\mu_2,\mu_5,\mu_3,\mu_6]$, $[\mu_1,\mu_7,\mu_4,\mu_8]$, $[\mu_2,\mu_5,\mu_4,\mu_8]$, $[\mu_2,\mu_5,\mu_4,\mu_8]$

(b) Groups $[\omega_1,\omega_7,\omega_2,\omega_5]$, $[\omega_1,\omega_7,\omega_3,\omega_6]$, $[\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_4,\omega_8]$ are added to candidate set of b_1

7. $<i_1,a_1>$ received

(a) a_1 can be placed to nodes $[\mu_9,\mu_{10},\mu_{11}]$, $[\mu_9,\mu_{10},\mu_{12}]$, $[\mu_{10},\mu_{11},\mu_{12}]$, $[\mu_9,\mu_{11},\mu_{12}]$

(b) Groups $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6,\omega_4,\omega_8]$, $[\omega_1,\omega_7,\omega_3,\omega_6,\omega_4,\omega_8]$ are added to candidate set of a_1

8. $<i_2,i_1>$ received

(a) i_1 can be placed to nodes $[\mu_1,\mu_7,\mu_2,\mu_5,\mu_3,\mu_6]$, $[\mu_1,\mu_7,\mu_2,\mu_5,\mu_4,\mu_8]$, $[\mu_2,\mu_5,\mu_3,\mu_6,\mu_4,\mu_8]$

(b) Groups $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6,\omega_4,\omega_8]$ are added to candidate set of i_1

9. $<i_2,y_1>$ received
   (a) y_1 can be placed to nodes $[\mu_9]$, $[\mu_{10}]$, $[\mu_{11}]$, $[P\mu_{12}]$
   (b) Groups $[\omega_1,\omega_7]$, $[\omega_2,\omega_5]$, $[\omega_3,\omega_6]$, $[\omega_4,\omega_8]$ are added to candidate set of y_1
   (c) $<y_1,a_1>$ pair added to visit queue
10. $<y_1,a_1>$ received
   (a) a_1 can be placed to nodes $[\mu_9]$, $[\mu_{10}]$, $[\mu_{11}]$, $[\mu_{12}]$
   (b) Groups $[\omega_1,\omega_7]$, $[\omega_2,\omega_5]$, $[\omega_3,\omega_6]$, $[\omega_4,\omega_8]$ are added to candidate set of a_1

After algorithm finishes its run, set of candidate core groups for each pattern is determined as follows.

- h_1: $[\omega_1,\omega_2,\omega_3,\omega_4,\omega_5,\omega_6,\omega_7,\omega_8]$
- b_1: $[\omega_1,\omega_7,\omega_2,\omega_5]$, $[\omega_1,\omega_7,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_3,\omega_6,\omega_4,\omega_8]$
- i_1: $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6,\omega_4,\omega_8]$, $[\omega_1,\omega_7,\omega_3,\omega_6,\omega_4,\omega_8]$
- i_2: $[\omega_1,\omega_7]$, $[\omega_2,\omega_5]$, $[\omega_3,\omega_6]$, $[\omega_4,\omega_8]$
- y_1: $[\omega_1,\omega_7]$, $[\omega_2,\omega_5]$, $[\omega_3,\omega_6]$, $[\omega_4,\omega_8]$
- a_1: $[\omega_1,\omega_7,\omega_2,\omega_5]$, $[\omega_1,\omega_7,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_3,\omega_6,\omega_4,\omega_8]$, $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_3,\omega_6]$, $[\omega_1,\omega_7,\omega_2,\omega_5,\omega_4,\omega_8]$, $[\omega_2,\omega_5,\omega_3,\omega_6,\omega_4,\omega_8]$, $[\omega_1,\omega_7,\omega_3,\omega_6,\omega_4,\omega_8]$, $[\omega_1,\omega_7]$, $[\omega_2,\omega_5]$, $[\omega_3,\omega_6]$, $[\omega_4,\omega_8]$

Using these set of candidate core groups, the following example scenario can be executed where each step represents a parallel sequence of method calls from the object of a dependency pattern to another object in the corresponding dependency pattern.

Step 1: $h_1 \rightarrow b_1 \rightarrow a_1$

Step 2: $h_1 \rightarrow i_2 \rightarrow y_1 \rightarrow a_1$

Step 3: $h_1 \rightarrow i_2 \rightarrow i_1 \rightarrow a_1$

Step 4: $h_1 \rightarrow i_1 \rightarrow b_1 \rightarrow a_1$

When this scenario executes, the allocation algorithm handles core affiliations as follows.

1. Initially all cores are marked as idle [ω_1:0,ω_2:0,ω_3:0,ω_4:0,ω_5:0,ω_6:0,ω_7:0,ω_8:0]
2. Step 1 kicks in
   (a) h_1 starts running. h_1 is scheduled to ω_1, ω_1 is marked in the affiliation list [ω_1:1,ω_2:0,ω_3:0,ω_4:0,ω_5:0,ω_6:0,ω_7:0,ω_8:0]
   (b) b_1 starts running. Objects of b_1 are scheduled to [ω_2,ω_5,ω_3,ω_6] consecutively. Affiliation list becomes [ω_1:1,ω_2:1,ω_3:1,ω_4:0,ω_5:1,ω_6:1,ω_7:0,ω_8:0]
   (c) a_1 starts running. a_1 is scheduled to the pool of(ω_2,ω_5,ω_3,ω_6). Affiliation list becomes [ω_1:1,ω_2:2,ω_3:2,ω_4:0,ω_5:2,ω_6:2,ω_7:0,ω_8:0]
3. Step 2 kicks in
   (a) h_1 continues to run. h_1 was scheduled to ω_1.
   (b) i_2 starts running. Objects of i_2 are scheduled to the pool of [ω_4,ω_8] consecutively. Affiliation list becomes [ω_1:1,ω_2:2,ω_3:2,ω_4:1,ω_5:2,ω_6:2,ω_7:0,ω_8:1]
   (c) y_1 starts running. y_1 is scheduled to ω_7. Affiliation list becomes [ω_1:1,ω_2:2,ω_3:2,ω_4:1,ω_5:2,ω_6:2,ω_7:1,ω_8:1]
   (d) Step 1's b_1 finishes. Affiliation list becomes [ω_1:1,ω_2:1,ω_3:1,ω_4:1,ω_5:1,ω_6:1,ω_7:1,ω_8:1]
   (e) a_1 starts running. a_1 is scheduled to the pool of (ω_1,ω_7). Affiliation list becomes [ω_1:2,ω_2:1,ω_3:1,ω_4:1,ω_5:1,ω_6:1,ω_7:2,ω_8:1]
   (f) Step 1's a_1 finishes. Affiliation list becomes [ω_1:1,ω_2:0,ω_3:0,ω_4:1,ω_5:0,ω_6:0,ω_7:1,ω_8:1]
4. Step 3 kicks in
   (a) h_1 continues to run. h_1 was scheduled to ω_1.
   (b) i_2 starts running. Objects of i_2 are scheduled to the pool of [ω_2,ω_5] consecutively. Affiliation list becomes [ω_1:1,ω_2:1,ω_3:0,ω_4:1,ω_5:1,ω_6:0,ω_7:1,ω_8:1]
   (c) i_1 starts running. i_1 is scheduled to the pool of (ω_1,ω_7,ω_2,ω_5,ω_3,ω_6). Affiliation list becomes [ω_1:2,ω_2:2,ω_3:1,ω_4:1,ω_5:2,ω_6:1,ω_7:2,ω_8:1]

(d) a_1 starts running. Assuming an object with an affiliation of ω_2 from the previous step calls a_1, a_1 is scheduled to the pool of (ω_2,ω_5). Affiliation list becomes [ω_1:2,ω_2:3,ω_3:1,ω_4:1,ω_5:3,ω_6:1,ω_7:2,ω_8:1]

5. Step 4 kicks in
   (a) h_1 continues to run. h_1 was scheduled to ω_1.
   (b) Step 2's i_2 finishes. Affiliation list becomes [ω_1:2,ω_2:3,ω_3:1,ω_4:0,ω_5:3,ω_6:1,ω_7:2,ω_8:0]
   (c) i_1 starts running. i_1 is scheduled to the pool of (ω_1,ω_7,ω_3,ω_6,ω_4,ω_8). Affiliation list becomes [ω_1:3,ω_2:3,ω_3:2,ω_4:1,ω_5:3,ω_6:2,ω_7:3,ω_8:1]
   (d) Step 3's i_2 finishes. Affiliation list becomes [ω_1:3,ω_2:2,ω_3:2,ω_4:1,ω_5:2,ω_6:2,ω_7:3,ω_8:1]
   (e) b_1 starts running. Assuming an object with an affiliation of ω_7 from the previous step calls b_1, b_1 is scheduled to the pool of (ω_1,ω_7,ω_4,ω_8). Affiliation list becomes [ω_1:4,ω_2:2,ω_3:2,ω_4:2,ω_5:2,ω_6:2,ω_7:4,ω_8:2]
   (f) Step 3's i_1 finishes. Affiliation list becomes [ω_1:3,ω_2:1,ω_3:1,ω_4:2,ω_5:1,ω_6:1,ω_7:3,ω_8:2]
   (g) a_1 starts running. Assuming an object with an affiliation of ω_1 from the previous step calls a_1, a_1 is scheduled to the pool of (ω_1,ω_7). Affiliation list becomes [ω_1:4,ω_2:1,ω_3:1,ω_4:2,ω_5:1,ω_6:1,ω_7:4,ω_8:2]
6. Rest of the objects finishes.

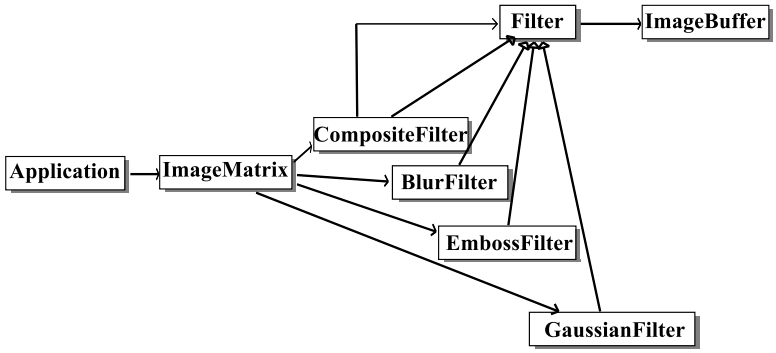
Please note that the core affiliation list held by the allocation algorithm is a superset of the actual core assignments at runtime. Graph matching algorithm has a complexity of $O(e)$ where e is the number of edges in the dependency graph.

4.2.3 Applying cache-aware dispatcher for a basic case study

In this section the cache-aware scheduling implementation will be applied to an image filtering application where a number of filters are applied to an image consecutively or at once using a composite filter. Firstly the software at hand will be parallelized using dependency patterns and then the proposed compile-time graph matching and runtime resource allocating algorithms will be applied to perform cache-aware scheduling on the software. Obtained results will be compared with linux' CFS and `O(1)` scheduling.

4.2.3.1 Case study software on image filtering

An image filtering software is chosen to be used as a case study in cache-aware scheduling. Image filtering software simply reads in an image as a matrix of gray levels for each pixel and convolves it with one or many filters defined in the software. Some filters may be chosen to be applied as a composite filter on the image. This feature is implemented by using a Composite pattern. The class diagram and the corresponding dependency diagram of the software can be found in Figure 4.12.



(a) Class diagram of image filtering software.

(b) Dependency pattern diagram of image filtering software in Figure 4.12a.

Figure 4.12: (a) presents class diagram of the case study and (b) presents dependency pattern diagram representing it.

It can be seen that Filter classes form a bridge from the hub class `ImageMatrix` to the authority `ImageMatrix`. During the implementation of the software Filter classes (as a bridge) impose a possible parallelization opportunity. On the other hand `CompositeFilter` forms a cycle which means a possible sequential behavior is present for this class. Care may need to be taken during the implementation of this class. Following classes reside in each of the dependency patterns in Figure 4.12b.

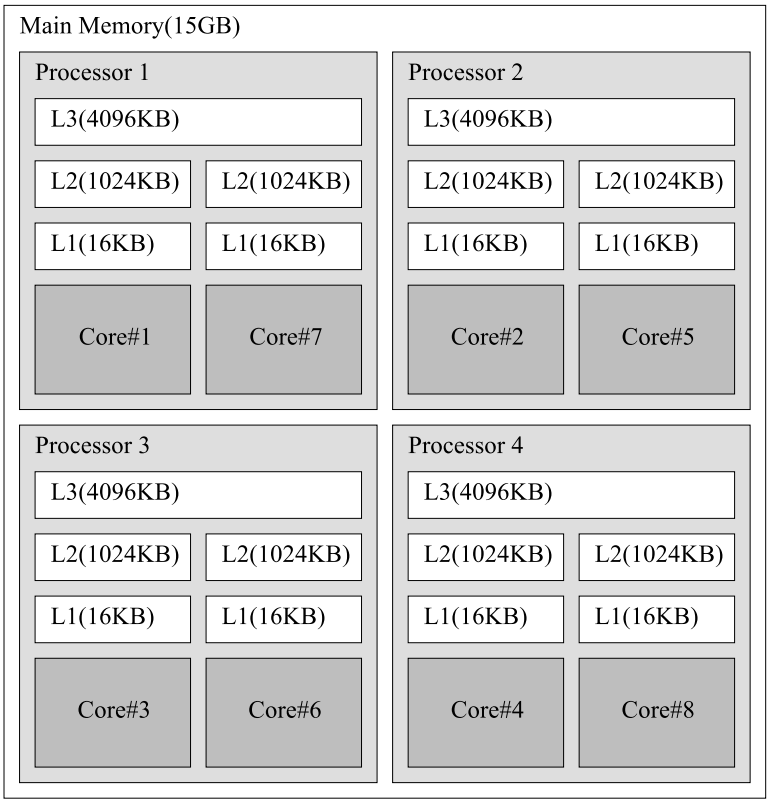
- H: `ImageMatrix`
- B: `CompositeFilter` `BlurFilter` `EmbossFilter` `GaussianFilter`
- C: `CompositeFilter`
- A: `ImageBuffer`

During the parallelization process, following the guidelines from the dependency patterns, each filter is programmed as a separate thread working on subsection of the image matrix. Each subsection is hold in an image buffer which is used by different filters consecutively. Also `CompositeFilter` class is parallelized since it applies each filter sequentially if implemented in its original form. When implemented this way it is possible to apply three different filters on the image in two different ways. Firstly each filter can be applied separately on the image, and each filter works parallel on the subsections of the image. Secondly filters can be applied as a composite filter where three filters applied consecutively on the subsections of the image in a parallel way. For instance if `BlurFilter` and `EmbossFilter` is to be applied on an image; for straightforward filtering `BlurFilter` is applied in a parallel way over the subsections of the image and `EmbossFilter` is applied afterwards. For composite filtering, image is decomposed into subsections first and `CompositeFilter` is applied where `BlurFilter` and `EmbossFilter` is applied consecutively for each subsection.

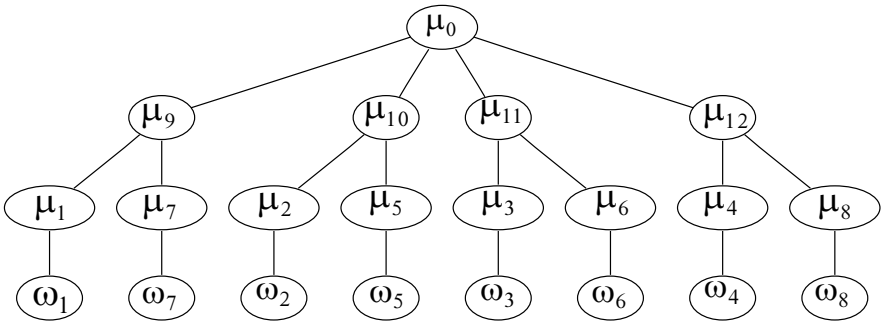
4.2.3.2 Experimental results

Experiments using cache-aware scheduling are performed in two separate processor architectures which can be seen in Figures 4.13 and 4.14 respectively. The processor in Figure 4.13 (referred to as TRW) is a 4 processor architecture where each processor holds 2 cores. The processor in Figure 4.14 (referred to as ZEB) is a 2 processor architecture where each processor holds 6 cores. TRW runs a Linux server with 2.6.32 kernel using a CFS scheduler and ZEB runs a Linux server with 2.6.18 kernel using `O(1)` scheduler.

During the experiments, image filtering software presented in the previous section has been slightly modified at each step to allow application of CAWS on different parallelization perspectives. Four different parallelization scenarios are applied which consist of applying composite filters on many subregions of a single image in parallel, applying many filters in parallel on a single image, applying many filters in parallel on multiple images and applying many filters on many subregions of multiple images in parallel.



(a) A sample 2 cored 4 processor processing unit.



(b) Graph representation of the processing unit in Figure 4.13a.

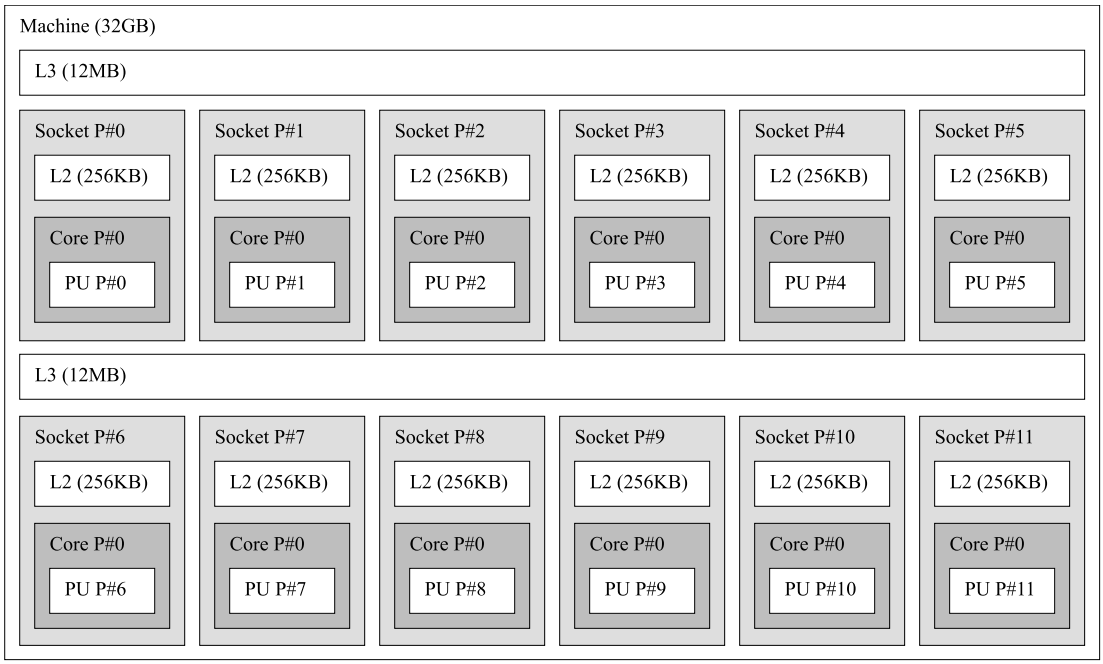
Figure 4.13: (a) presents an example processor and (b) presents the processor-memory hierarchy graph representing it.

In all the plots presented below y-axis represents normalized runtime performance where normalization is performed by calculating for each experiment.

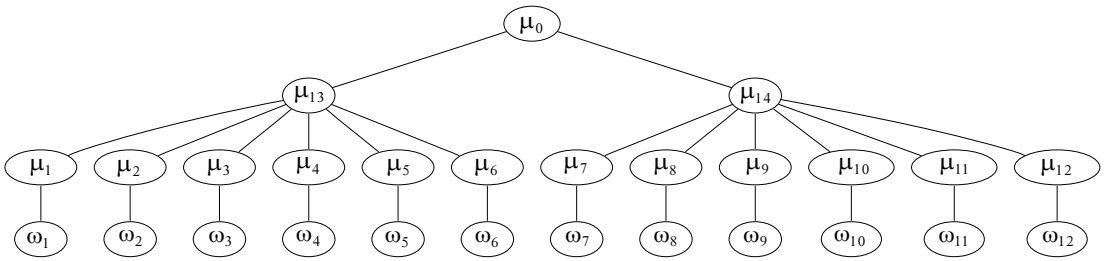
$$\rho_i = \frac{1}{T_i} \tag{4.6}$$

$$\rho_n = \frac{\rho_i}{\rho_i^{(best)}} \times 100 \tag{4.7}$$

In Equations **(4.6)** and **(4.7)**, T_i represents avarage running time for each case, ρ_i represents performance of each case and $\rho_i^{(best)}$ is the best performance(lowest T_i, highest ρ_i) among all measurements for the plot at hand. Multiplicating the result



(a) A sample 6 cored 2 processor processing unit.



(b) Graph representation of the processing unit in Figure 4.14a.

Figure 4.14: (a) presents an example processing unit and (b) presents the processor-memory hierarchy graph representing it.

by 100 enables to easily read the performance differences between measurements with terms of percentage. Confidence intervals of all the experimental results are obtained by repeating experiments until results converge. A repetition of 25 times were enough during the experiments.

Applying composite filters on many subregions of an image in parallel

For the first scenario in Figure 4.15 different `CompositeFilter` objects are created and run in parallel over an image kept in an `ImageMatrix` object. The image is decomposed into many subparts each hold in a different `ImageBuffer` object, so that each composite filter instance can work on the image in parallel. For this scenario filters inside a composite object run sequentially and reuse the corresponding

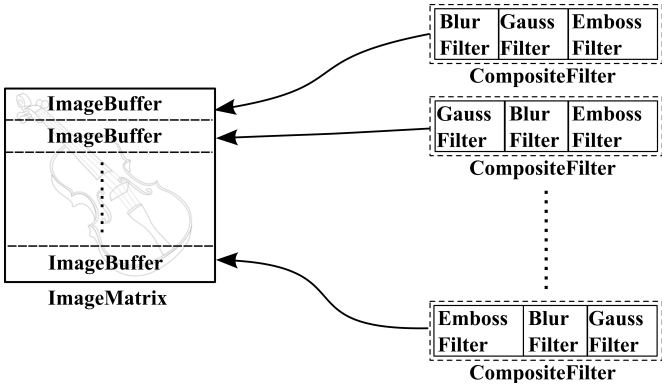


Figure 4.15: Applying composite filters on many subregions of a single image in parallel.

`ImageBuffer` after it has been convolved by the predecessor of the filter. This way each subsections of the image are reused by the filters of a `CompositeFilter`.

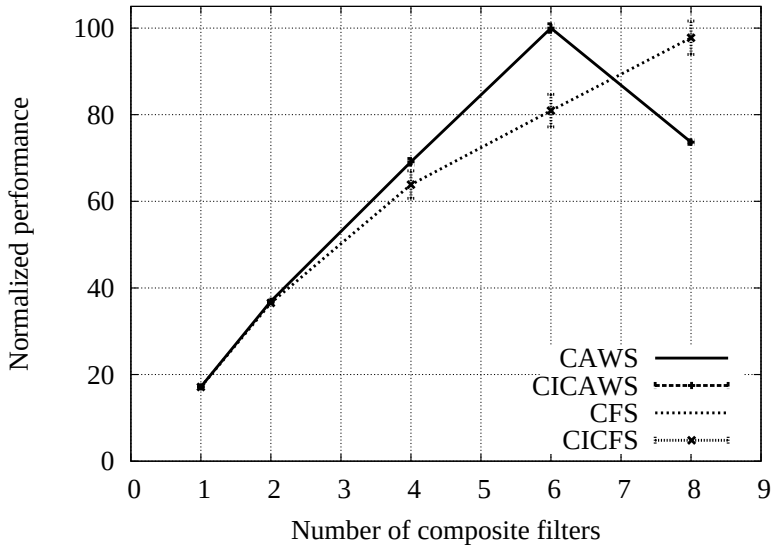


Figure 4.16: TRW results for applying composite filters on many subregions of a single image in parallel.

In Figure 4.16, performance results of image filtering with respect to the increase of parallelized regions inside the image is presented for TRW. The performance of both CFS and CAWS increases with the number of parallelized parts. CAWS outperformed CFS until the number of parallelized regions reaches the number of cores in the system. In Figure 4.17 the peak performance diffrence versus CFS is not as much as the former case, however an improvement can be seen as the number of parallel parts goes beyond 6(which is the number of cores that share a common cache).

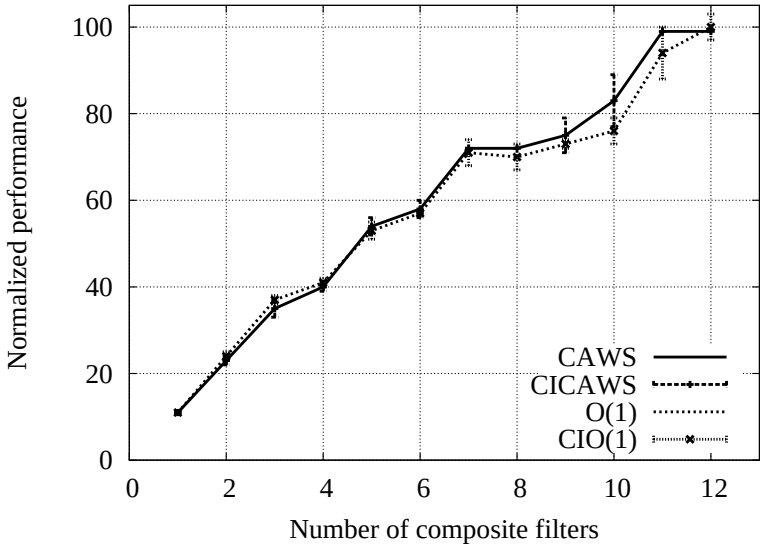


Figure 4.17: ZEB results for applying composite filters on many subregions of a single image in parallel.

In all of the experiments presented in this section CAWS competed successfully with CFS until the number of parallelized regions reached the number of cores. This happens because the CAWS implementation used in the experiments doesn't migrate threads once it affiliates a thread with a core. Because of this after the number of threads reaches the number of cores the affiliation of threads that waits in processor queues should be updated frequently considering the workload of the cores. However to make such a decision cost of migrating the thread should be compared with the cost of cache misses migration is going to trigger. Making this kind of decisions is a subject which is out of this dissertation's scope so the numbers until the number of threads reaches the number of cores is presented in results.

Applying many filters in parallel on a single image

In Figure 4.18 second scenario is depicted where many different *Filter* instances are applied on an image in parallel. This time `ImageMatrix` has only one `ImageBuffer` which keeps the entire image. On the other hand unlike the first scenario this time every filter gets the same `ImageBuffer` but produces its own copy by convolving on it. In this example, entire image is reused by different filter instances in parallel.

In Figure 4.19, performance results of image filtering with respect to the number of filters is presented for TRW. CAWS outperformed CFS until the number of parallelized

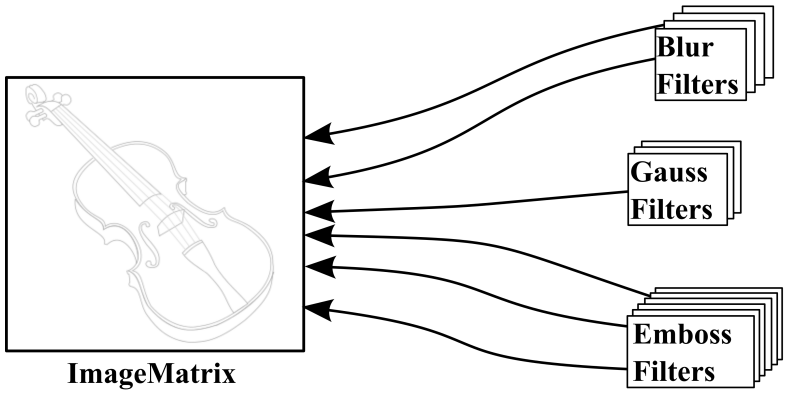


Figure 4.18: Applying many filters in parallel on a single image.

filters reached the number of cores in the system. At the end of the experiments, results show that the improvement performed by CAWS increases up to ∼20%.

In Figure 4.20 similar results can be seen for ZEB's `O(1)` where performance improvement reached 20%. It can be seen that performance peaks emerged when the number of filters(parallel working threads) reaches the factors of number of cores that share the same level cache(6 and 12 in our case). When the number of data sharing software components spans the number of cores using common cache performance drops substantially.

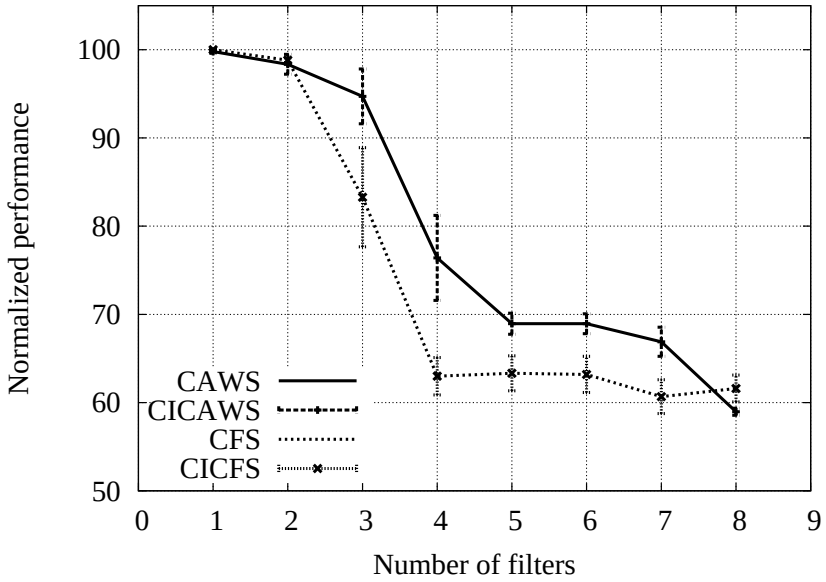


Figure 4.19: TRW results for applying many filters in parallel on a single image.

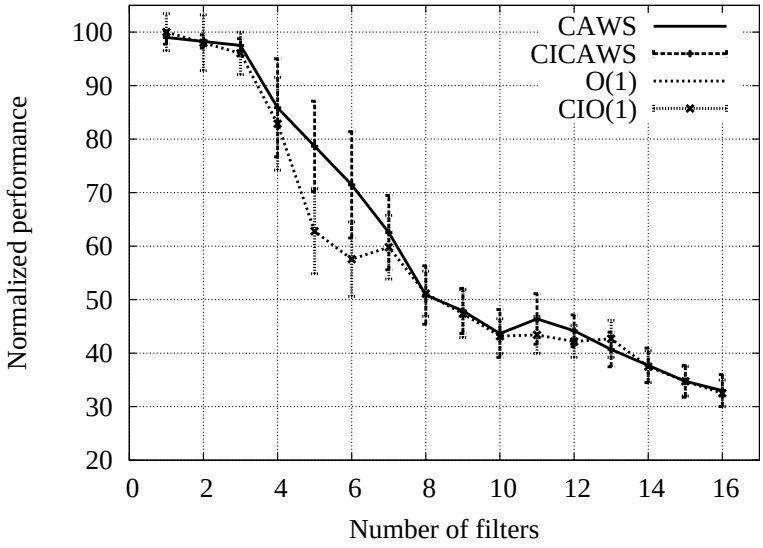


Figure 4.20: ZEB results for applying many filters in parallel on a single image.

Applying many filters in parallel on multiple images

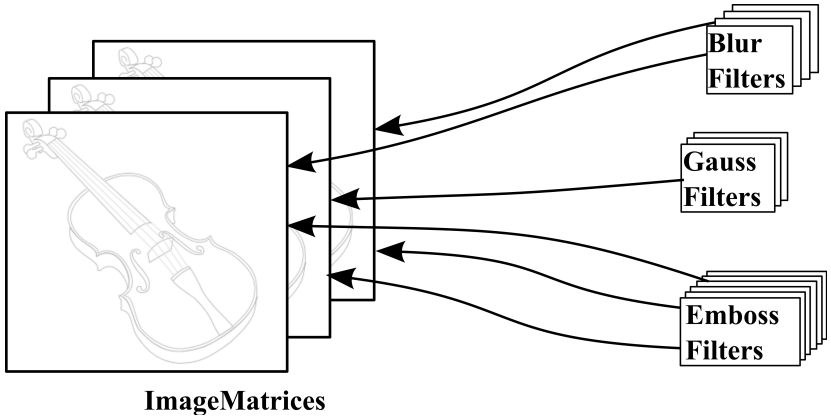


Figure 4.21: Applying many filters in parallel on multiple images.

To make the previous scenario more realistic and increase the importance of thread distribution multiple images are used for the scenario in Figure 4.21. When two or three images are used instead of one, number of cache misses starts to increase if the filters are placed randomly to the cores.

In Figures 4.22 and 4.23 results from two different perspectives are presented for TRW. In Figure 4.22 the performance results of CFS with respect to varying number of filters on two different images is presented. In Figure 4.23 the number of filters is constant where the number of images varies. For both of the cases performance improvement has reached up to around 20%.

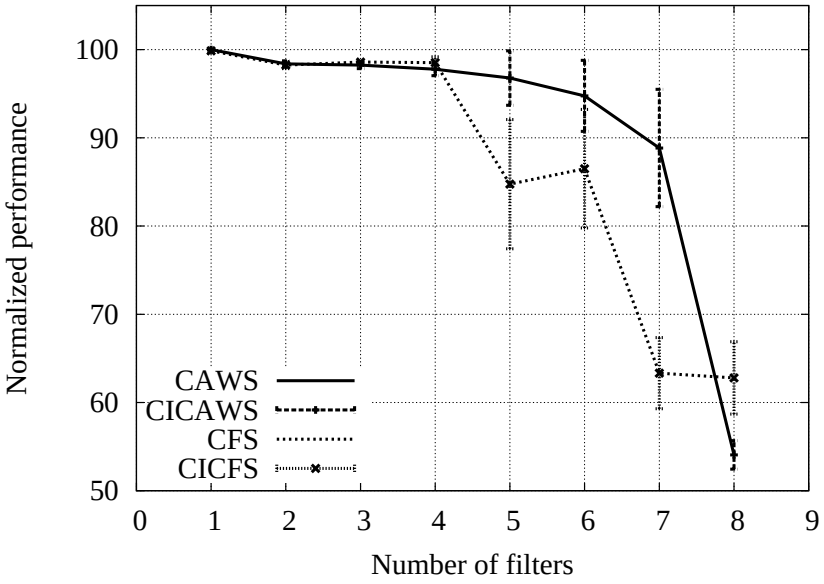


Figure 4.22: TRW results for applying many filters in parallel on two images.

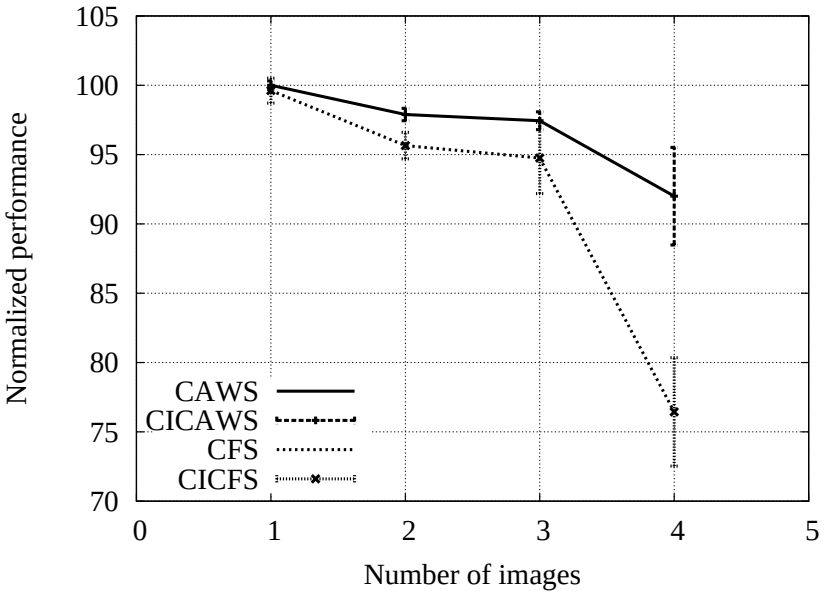


Figure 4.23: TRW results for applying many filters in parallel on many images.

In Figures 4.24 and 4.25 results using two and three different images in ZEB's `O(1)` are presented successively. For both cases performance improvement has reached up to around 30%. For parallel filters on two images performance of CAWS started to fall behind `O(1)` after 10 filters because each image gets 5 filters for this case allowing CAWS to place 6 objects(1 image 5 filters) to the neighboring cores ZEB. For parallel filters on three images CAWS started to be outperformed after 9 filters since the number of total objects in the system exceeds the number of cores.

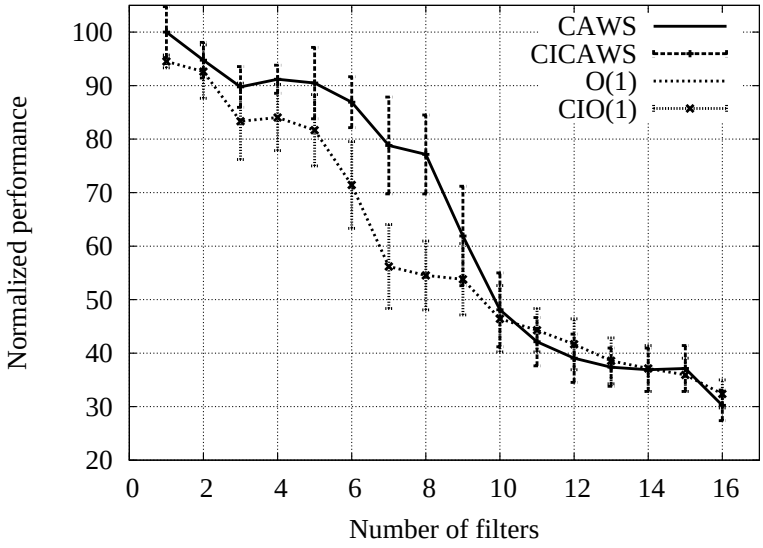


Figure 4.24: ZEB results for applying many filters in parallel on two images.

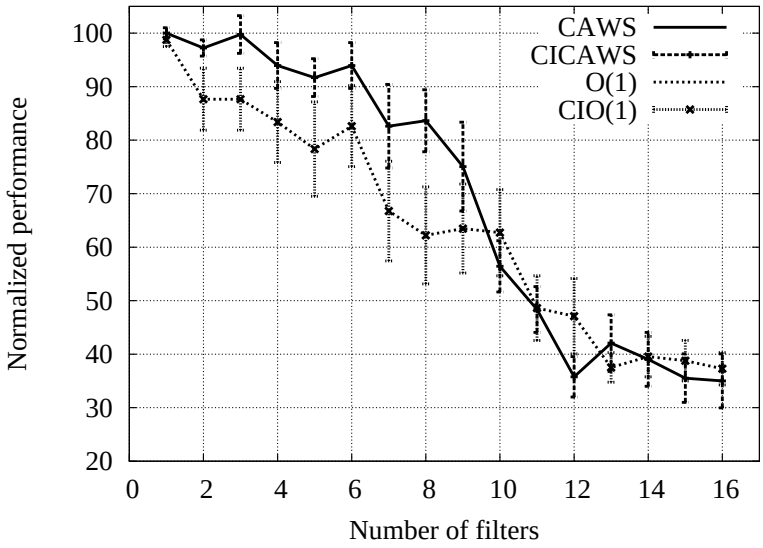


Figure 4.25: ZEB results for applying many filters in parallel on three images.

Applying many filters on subregions of multiple images in parallel

For the last scenario in Figure 4.26 multiple images are even decomposed into many different subregions, each being held by a different `ImageBuffer` object. In this scenario the number of parallelized parts are a lot more than the previous scenarios increasing the chance of CAWS dispatcher to affiliate filters that work on different `ImageBuffers` (in other words filters that work on different data) resulting increased miss rates and degrading performance. It is not meaningful to conduct such an experiment with double cores sharing a cache so the experiments are only run for ZEB for this scenario.

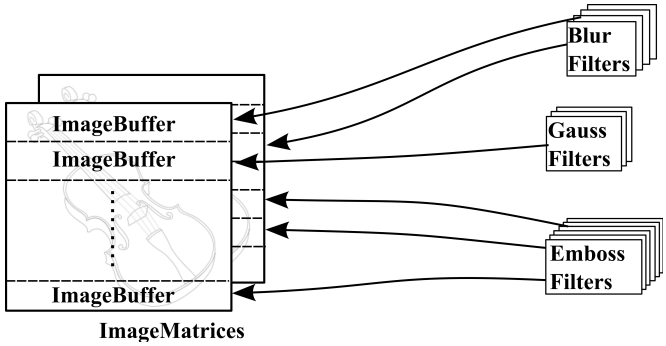


Figure 4.26: Applying many filters on many subregions of multiple images in parallel.

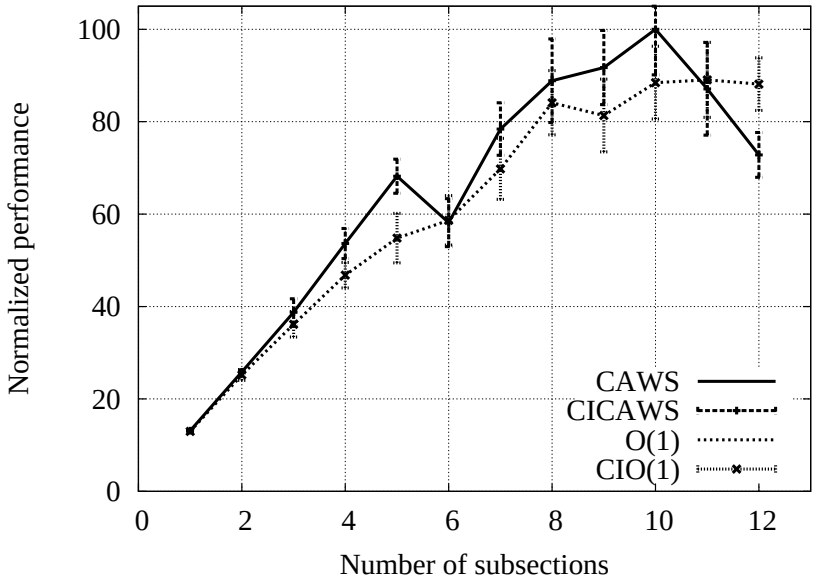


Figure 4.27: ZEB results for applying many filters on many subregions of an image in parallel.

In Figures 4.27 and 4.28, results using one and two different images consisting of many subregions in ZEB are presented successively. In the results x-axis present the number of parallelized parts which is the total number of subsections inside all the images in the system. For both cases performance improvement has reached up to around 20%. In Figure 4.27 after the number of subsections reach over 6 CAWS starts to place subsections to non-neighboring cores which result in smaller performance improvements after 6 subsections. In Figure 4.28 it can be seen that CAWS continues to compete with `O(1)` until 10 cumulative subsections to be filterred present in the system which results in two images and 5 filter objects convolving subsections of the image at a given time. In this scenario each image object is assigned to another shared cache maximizing cache reuse.

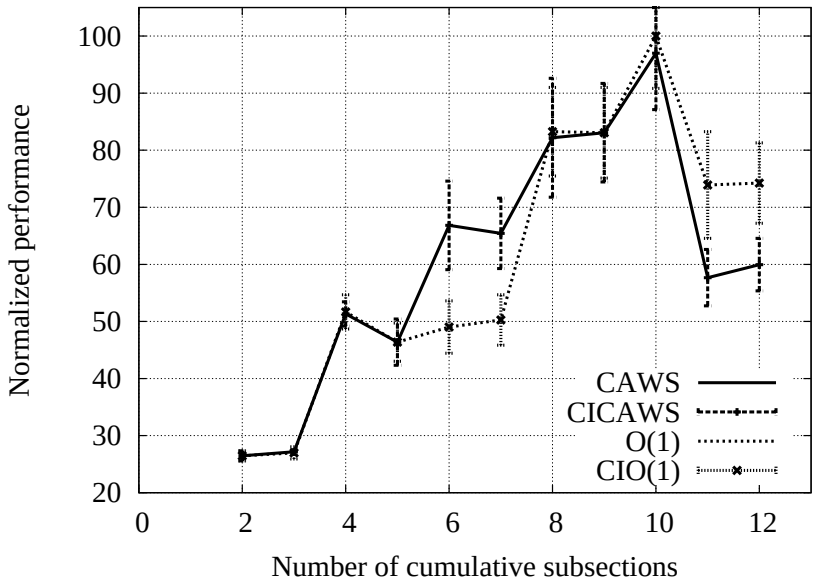


Figure 4.28: ZEB results for applying many filters on many subregions of two images in parallel.

As a result of all the experiments above, it can be seen that CAWS provides a maximum performance improvement of 20% in almost all of the cases. Performance improvement has even reached up to 30% for two scenarios. On average CAWS provides $\sim$10% performance improvement if thread migration is not required at runtime.

4.3 Summary and Conclusions

The studies on cache-aware scheduling presented in this chapter show that considering shared data during scheduling increases the scheduling performance when multicore processors are used. It is important to utilize shared data among software components in guiding the scheduling process, even if it is not always possible to make accurate predictions on data sharing among software components before the system is run.

The approach presented in Section 4.1 uses software models to reason about data sharing among the classes of a software. Experimentations on three different commonly used software design patterns to consider the effect of cache-aware scheduling. Promising results are obtained to apply a model-based approach on larger software considering the three important factors (parallelization, data sharing and resource utilization) that effect the overall performance of the system in the presented case studies. Beside its positive effects on scheduling performance, using

a model driven approach may lead to reason about software design for various core organizations that processors can include in the future.

In Section 4.2 a dispatcher implementation is presented that uses CAWS principles to match the dependency graph of a software with the memory-core hierarchy graph of a processor. The results obtained by applying dispatcher on an image filtering case study outperformed Linux' CFS/`O(1)` with a rate up to 30% and showed promising results for CAWS. The improvement continued until the number of threads has reached the number of cores in the system since experimented CAWS implementation doesn't perform thread migration which is left as a future study for the dispatcher.

As future studies the presented model based dispatcher can be improved based on the lessons obtained from the experiments presented in the last section. By using a model driven dispatcher and cache aware scheduling methodology it can be possible to reason about parallelization and data sharing during the early design stage of software development. Moreover it can be possible to steer the design/development process to produce more competing designs for parallelization when different processor architectures are used.

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

At a first glance, model based analysis and runtime performance of a software seem to stand almost at two distant phases of the software life cycle. However, as the recent studies show, the decisions made at earlier stages of software development has the most serious effects compared to latter stages. This thesis encourages that parallelization can be seen as one of these decisions. However, it is more difficult to make such decisions since parallelism is harder to detect at the earlier stages of software life. On the other hand using model based analysis makes it possible to develop more efficient parallelization solutions at earlier stages of software development. It is harder to detect and utilize such parallelization opportunities at development phase. Following studies are achieved in thesis to improve software quality for multicore systems.

In Chapter 3 dependency patterns and their occurrence in class diagrams are presented which played key role in thesis studies. Utilization of dependency patterns in parallelization and synchronization efforts are shown using a case study(Jikes). Later, some more detailed examples are also presented on different object oriented software. In those examples more detailed properties of dependency patterns are analysed using software metrics. A metric set is defined in that purpose and the set of properties that can be covered using defined set of metrics are presented. By using the proposed set of metrics on dependency patterns found in different object oriented software, properties of dependency pattern instances during parallelization process are analyzed. Finally, using clustering techniques in exploring dependency patterns inside class diagrams is discussed and an improvement is proposed for this process. Some of the obtained schemes almost doubled the clustering performance for dependency patterns and a noticeable improvement is obtained for the most of them. Moreover by using the proposed technique, overlapping structures that cannot be found using conventional clustering methods can be detected as well.

Chapter 4 presents improvement of object oriented software scheduling by analyzing possible data sharing among software components using dependency patterns. Dependency patterns capture the possible coupling between classes of software at runtime. The proposed technique uses this feature to reason about common data usage among software classes and place related classes' objects to the cores that share the same level cache. Of course this placement policy is effected by the number of objects that can be produced at runtime and the way a group of related objects distributed regarding the architecture of the processor at hand. Firstly, to examine the applicability of the cache-aware scheduling, technique is applied to basic implementations of software design patterns and promising results are obtained. Later in this chapter an example implementation of an object dispatcher is presented that uses cache-aware scheduling principles. Final results showed that by applying the proposed technique, Linux' CFS/`O(1)` scheduler performance can be improved up to 25%.

The results show that it is possible to detect and utilize such implicit parallelization decisions by analyzing class diagrams. Moreover it is even possible to fine tune scheduling of object oriented software using the results of such analysis. As the main objective of parallelization efforts, the results in the thesis showed that it is possible to obtain up to a maximum 25% performance improvement. In avarage, proposed technique resulted around 10% performance improvement. But almost as important as performance gains, having insight about other aspects of parallel software quality, like synchronization, is another outcome of proposed methodology. Last but not least, applying model based analysis and pattern based solutions makes it easier to maintain software quality during refactoring for parallelization.

As a final evaluation, the proposed improvements for parallelization presented in this thesis are one of the early studies on model driven software refactoring for parallel development that also considers quality based properties of modern software. Thesis studies cover a wide range of topics from software metrics to scheduling and present original and influential ideas over a complete range of properties on discovering implicit parallelism in object oriented software.

5.2 Future Work

The future works are as follows:

- All the analysis presented in the thesis studies are performed based solely on static models of software. Although this situation brings vagueness to the results of the analysis, using dynamic models also has its own difficulties discussed in Chapter 3. However results from the static analysis may be enhanced using information from the dynamic diagrams and dynamic analysis of software. Especially for scheduling of object oriented software feedback can be obtained from dynamic analysis of related software.
- Dispatcher system proposed in Chapter 4 makes affinity settings for threads at creation time. This causes the proposed system to fail in load balancing after the number of parallel objects exceeds the number of cores because dispatcher doesn't update core affinities once threads are created. By introducing a dynamic affinity updating system and migrating the threads to appropriate idle cores at runtime the performance of the dispatcher can be improved. However this improvement brings a lot of difficulties like the need of estimating the cost of migrating the object over waiting for its completion which exceeds the scope of this thesis.
- Dependency diagrams are not specific only to parallel analysis. Instead they are closely related with graph cluster based structures and can be applied not only in many different areas of software engineering but also many areas of computer science(like web mining) as well where inter-graph relations pose important structures.
- During thesis studies one of the major obstacles were finding a variety of different software that were designed in an object oriented way, and parallelized neatly. In order to observe different distribution and parallelization techniques a software simulator can be very handy. As a future work, for simulating various different software model runs before the software is implemented, a software simulator(like network simulators) can be implemented.
- There exist many multicore processor simulators in the literature, but most of them includes very detailed configuration options to simulate the hardware in a detailed

way. Another need to work on software design for multicore systems is a multicore processor simulator focused only on core-memory hierarchy of the processor with a simple configuration interface.

- Based on the software and multicore processor simulators mentioned in the last two items, the vision based on the thesis studies is being able to rapidly model the software and processor architectures to reason about necessary modifications on the software model as well as minimum acceptable needs that the processor shall serve to obtain specific performance requirements from software. A framework like this may hopefully form a bridge between respectively complicated hardware and software design world in the future.

REFERENCES

[1] **Comer, D.E.**, **Gries, D.**, **Mulder, M.C.**, **Tucker, A.**, **Turner, A.J. and Young, P.R.**, 1989. Computing as a Discipline, *Communications of the ACM*, **32**, 9–23.

[2] **Flynn, M.J.**, 1972. Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, **21**, 948–960.

[3] **Joch, A.** Chip Multiprocessing, *<http://www.computerworld.com/s/article/54343/Chip_Multiprocessing>*, accessed at 11.04.2011.

[4] **Bik, A.J.C. and Gannon, D.B.**, 1997. Automatically Exploiting Implicit Parallelism in Java, *Concurrency, Practice and Experience*, pp. 579–619.

[5] **Oliver, J.**, **Guitart, J.**, **Ayguadé, E.**, **Navarro, N. and Torres, J.**, 2001. Strategies for the efficient exploitation of loop-level parallelism in Java, *Concurrency and Computation: Practice and Experience*, **13**, 663–680.

[6] **Bull, J. and Kambites, M.**, 2000. JOMP—an OpenMP-like interface for Java, *Proceedings of the Conference on Java Grande*, ACM, pp. 44–53.

[7] **Felber, P.**, 2003. Semi-automatic Parallelization of Java Applications, *On The Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 1369–1383.

[8] **Du, J.**, **Chen, D. and Xie, L.**, 1999. JAPS: an automatic parallelizing system based on Java, *Science in China Series E: Technological Sciences*, **42**, 396–406.

[9] **Yu, M.**, **Guo, M.**, **Pan, Y.**, **Zang, W. and Xie, L.**, 2002. JAPS-II: A Source to Source Parallelizing Compiler for Java, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 1*, PDPTA '02, CSREA Press, pp. 164–170.

[10] **Guitart, J.**, **Martorell, X.**, **Torres, J. and Ayguadé, E.**, 2001. Efficient Execution of Parallel Java Applications, *3rd Annual Workshop on Java for High Performance Computing (part of the 15th ACM International Conference on Supercomputing)*, ICS'01, pp. 31–35.

[11] **Chan, B. and Abdelrahman, T.S.**, 2004. Run-Time Support for the Automatic Parallelization of Java Programs, *The Journal of Supercomputing*, **28**(1), 91–117.

[12] **Halvorsen, O.**, **Runde, R.K. and Haugen, Ø.**, 2007. Time Exceptions in Sequence Diagrams, *Models in Software Engineering*, volume 4364 of

Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 131–142.

[13] **Kaveh, N.**, 2001. Using Model Checking to Detect Deadlocks in Distributed Object Systems, *Revised Papers from the Second International Workshop on Engineering Distributed Objects.*, EDO'00, Springer-Verlag, London, UK, pp. 116–128.

[14] **Mitchell, B.**, 2008. Characterizing Communication Channel Deadlocks in Sequence Diagrams, *IEEE Transactions on Software Engineering*, **34**(3), 305–320.

[15] **Eshuis, R.**, 2006. Symbolic model checking of UML activity diagrams, *Transactions on Software Engineering and Methodology*, **15**(1), 1–38.

[16] **Latella, D.**, **Majzik, I. and Massink, M.**, 1999. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker, *Formal Aspects of Computing*, **11**, 637–664.

[17] **Holzmann, G.J.**, 1991. *Design and validation of computer protocols*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[18] **Holzmann, G.J.**, 1997. The Model Checker SPIN, *Software Engineering*, **23**(5), 279–295.

[19] **Newman, E. and Greenhouse, A.**, 2001. Annotation-based Diagrams for Shared-Data Concurrency, *Workshop on Concurrency Issues in UML at the Fourth International Conference on the Unified Modeling Language*.

[20] **Mehner, K. and Wagner, A.**, 2000. Visualizing the Synchronization of Java-Threads with UML, *Proceedings of the IEEE International Symposium on Visual Languages*, VL'00, IEEE Computer Society, Washington, DC, USA, p. 199.

[21] **Konrad, S.**, **Campbell, L.A. and Cheng, B.H.C.**, 2004. Automated Analysis of Timing Information in UML Diagrams, *Proceedings of the Nineteenth IEEE international conference on Automated software engineering*, ASE'04, IEEE Computer Society, Washington, DC, USA, pp. 350–353.

[22] **Das, D.**, **Chakrabarti, P.P. and Kumar, R.**, 2007. Functional verification of task partitioning for multiprocessor embedded systems, *Transactions on Design Automation of Electronic Systems*, **12**(4), 44.

[23] **Davies, J. and Crichton, C.**, 2003. Concurrency and Refinement in the Unified Modeling Language, *Formal Aspects of Computing*, **15**(2), 118–145.

[24] **Edwards, D.**, **Simmons, S. and Kearns, P.**, 2004. Graphical Limits of Concurrency, *Neural Parallel And Scientific Computations*, **12**, 219–232.

[25] **Plasil, F. and Mencl, V.**, 2003. Getting 'Whole Picture' Behavior In A Use Case Model, *Journal of Integrated Design and Process Science*, **7**(4), 63–79.

[26] **Sethumadhavan, S.**, **Arora, N.**, **Ganapathi, R.B.**, **Demme, J. and Kaiser, G.E.**, 2009. COMPASS: A Community-driven Parallelization Advisor for Sequential Software, *Proceedings of the ICSE Workshop on Multicore Software Engineering*, IWMSE'09, IEEE Computer Society, Washington, DC, USA, pp. 41–48.

[27] **Erraguntla, R. and Carver, D.L.**, 1998. Migration of sequential systems to parallel environments by reverse engineering, *Information & Software Technology*, **40**(7), 369–380.

[28] **Kim, S.**, **Ch, D. and Solihin, Y.**, 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture, *In IEEE PACT*, pp. 111–122.

[29] **Tam, D.**, **Azimi, R.**, **Soares, L. and Stumm, M.**, 2007. Managing Shared L2 Caches on Multicore Systems in Software, *Workshop on the Interaction between Operating Systems and Computer Architecture*.

[30] **Tam, D.**, **Azimi, R. and Stumm, M.**, 2007. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors, *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ACM, New York, NY, USA, pp. 47–58.

[31] **Merkel, A. and Bellosa, F.**, 2008. Memory-aware scheduling for energy efficiency on multicore processors, *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, USENIX Association, Berkeley, CA, USA, pp. 1–1.

[32] **Ha, J.**, **Arnold, M.**, **Blackburn, S.M. and McKinley, K.S.**, 2009. A concurrent dynamic analysis framework for multicore hardware, *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, ACM, New York, NY, USA, pp. 155–174.

[33] **Zhou, B.**, **Qiao, J. and kuan Lin, S.**, 2009. Research on Dynamic Cache Distribution Scheduling Algorithm on Multi-Core Processors, *E-Business and Information System Security, 2009. EBISS '09. International Conference on*, pp. 1 –4.

[34] **Zangerl, T.**, 2008. Optimisation: Operating System Scheduling on multi-core architectures, <*http://tzangerl.net/doc/MulticoreScheduling.pdf*>, accessed at 11.04.2011.

[35] **Siddha, S.**, 2007. Multi-core and Linux Kernel, <*http://software.intel.com/sites/oss/pdfs/mclinux.pdf*>, accessed at 11.04.2011.

[36] **Microsoft**, 2011. MSDN Section on Windows Scheduling, <*http://msdn.microsoft.com/en-us/library/ms685096(v=vs.85).aspx*>, accessed at 11.04.2011.

[37] **Oracle**, 2010. Solaris 11 Programming Interfaces Guide, <*http://download.oracle.com/docs/cd/E19963-01/html/821-1602/psched-23069.html*>, accessed at 11.04.2011.

[38] **Boyd-Wickizer, S.**, **Morris, R. and Kaashoek, M.F.**, 2009. Reinventing Scheduling for Multicore Systems, *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verità, Switzerland.

[39] **Xue, L.**, **Kandemir, M.T.**, **Chen, G.**, **Li, F.**, **Ozturk, O.**, **Ramanarayanan, R. and Vaidyanathan, B.**, 2007. Locality-Aware Distributed Loop Scheduling for Chip Multiprocessors, *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, IEEE Computer Society, Washington, DC, USA, pp. 251–258.

[40] **Fedorova, R.**, **Seltzer, M. and , M.D.S.**, 2006. Cache-fair thread scheduling for multicore processors, *Technical Report*, Harvard University.

[41] **Koziris, N.**, **Romesis, M.**, **Tsanakas, P. and Papakonstantinou, G.**, 2000. An efficient algorithm for the physical mapping of clustered task graphs onto multiprocessor architectures, *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, pp. 406 –413.

[42] **Trifunovic, A. and Knottenbelt, W.J.**, 2006. A General Graph Model for Representing Exact Communication Volume in Parallel Sparse Matrix-Vector Multiplication, *ISCIS*, pp. 813–824.

[43] **Roig, C.**, **Ripoll, A. and Guirado, F.**, 2007. A New Task Graph Model for Mapping Message Passing Applications, *Parallel and Distributed Systems, IEEE Transactions on*, **18**(12), 1740 –1753.

[44] **Chen, G.**, **Li, F.**, **Son, S. and Kandemir, M.**, 2008. Application mapping for chip multiprocessors, *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 620 –625.

[45] **Xie, Y. and Loh, G.H.**, 2009. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches, *In Proc. of the 36th Intl. Symp. on Computer Architecture*, pp. 174–183.

[46] **Valiant, L.G.**, 2008. A Bridging Model for Multi-core Computing, *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pp. 13–28.

[47] **Kumar, V. and Delgrande, J.**, 2009. Optimal Multicore Scheduling: An Application of ASP Techniques, *LPNMR '09: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, Springer-Verlag, Berlin, Heidelberg, pp. 604–609.

[48] **Yau, S.**, **Jia, X.**, **Bae, D.**, **Chidambaram, M. and Oh, G.**, 1991. Using Model Checking to Detect Deadlocks in Distributed Object Systems, *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference.*, COMPSAC'91, pp. 453–458.

[49] **Yau, S.**, **Bae, D. and Pour, G.**, 1992. A partitioning approach for object-oriented software development for parallel processing systems, *Proceedings of the Sixteenth Annual International Computer Software and Applications Conference.*, COMPSAC'92, pp. 251–256.

[50] **Li, X. and Lilius, J.**, 1999. Timing analysis of UML sequence diagrams, *Proceedings of the Second International Conference on The Unified Modeling Language. Beyond the Standard.*, Springer, Fort Collins, CO, USA, pp. 661–674.

[51] **Li, X.**, **Meng, C.**, **Yu, P.**, **Jianhua, Z. and Guoliang, Z.**, 2001. Timing Analysis of UML Activity Diagrams, *Proceedings of the Fourth International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Springer-Verlag, London, UK, pp. 62–75.

[52] **Engels, G.**, **Küuster, J. and Groenwegen, L.**, 2002. Consistent Interaction Of Software Components, *Journal of Integrated Design and Process Science*, **6**(4), 2–22.

[53] **Giese, H.**, **Klein, F. and Burmester, S.**, 2005. Pattern Synthesis from Multiple Scenarios for Parameterized Real-Time UML Models, *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 193–211.

[54] **Seiter, L.**, **Palsberg, J. and Lieberherr, K.**, 1998. Evolution of Object Behavior Using Context Relations, *IEEE Transactions on Software Engineering*, **24**, 79–92.

[55] **Cazzola, W.**, **Ghoneim, A. and Saake, G.**, 2002. Reflective Analysis and Design for Adapting Object Run-Time Behavior, *Proceedings of the Eighth International Conference on Object-Oriented Information Systems*, OOIS'02, Springer-Verlag, London, UK, pp. 242–254.

[56] **Gamma, B.**, **Helm, R.**, **Johnson, R. and Vlissides, J.**, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

[57] **IBM**. Jikes, <*http://jikes.sourceforge.net/*>, accessed at 01.07.2011.

[58] **Chatzigeorgiou, A.**, **Tsantalis, N. and Stephanides, G.**, 2006. Application of graph theory to OO software engineering, *Proceedings of the International Workshop on interdisciplinary software engineering research.*, WISER'06, ACM, New York, NY, USA, pp. 29–36.

[59] **Karypis, G. and Kumar, V.**, 1995. A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal on Scientific Computing*, **20**, 359–392.

[60] **Ng, A.**, **Jordan, M. and Weiss, Y.**, 2001. On spectral clustering: Analysis and an algorithm, *Advances in Neural Information Processing Systems*, MIT Press, pp. 849–856.

[61] **van Dongen, S.**, 2000. Graph Clustering by Flow Simulation, *Ph.D. thesis*, University of Utrecht, The Netherlands.

[62] **Ovatman, T. and Buzluca, F.**, 2008. Investigating software design pattern behavior in multiprocessor systems: A case study on observer,

23rd International Symposium on Computer and Information Sciences, ISCIS'08, pp. 1 –4.

[63] **Ovatman, T. and Buzluca, F.**, 2009. Software Design Pattern Behavior in Shared Memory Multiprocessor Systems, *International Conference on Computational Intelligence and Software Engineering*, CiSE'09, pp. 1 –4.

[64] **Chidamber, S.R. and Kemerer, C.F.**, 1994. A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, **20**(6), 476–493.

[65] **Brito e Abreu, F. and Carapuca, R.**, 1994. Object-Oriented Software Engineering: Measuring and Controlling the Development Process, *Proc. Int'l Conf. Software Quality*.

[66] **Harrison, R.**, **Counsell, S. and Nithi, R.**, 1998. An Evaluation of the MOOD Set of Object-Oriented Software Metrics, *IEEE Transactions on Software Engineering*, **24**, 491–496.

[67] **Bansiya, J. and Davis, C.G.**, 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment, *IEEE Transactions on Software Engineering*, **28**(1), 4–17.

[68] **Briand, L.C.**, **Morasca, S. and Basili, V.R.**, 1996. Property-Based Software Engineering Measurement, *IEEE Trans. Softw. Eng.*, **22**(1), 68–86.

[69] **Briand, L.**, **Arisholm, E.**, **Counsell, S.**, **Houdek, F. and Thévenod-Fosse, P.**, 1999. Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of The Art and Future Directions, *Empirical Software Engineering*, **4**, 387–404.

[70] **McCabe, T.J.**, 1976. A Complexity Metric, *IEEE Transactions on Software Engineering*, **2**.

[71] **Li, Z.**, **Mills, P.H. and Reif, J.H.**, 1989. Models and Resource Metrics for Parallel and Distributed Computation, *Proc. 28th Annual Hawaii International Conference on System Sciences*, pp. 133–143.

[72] **Hollingsworth, J.K. and Miller, B.P.**, 1992. Parallel program performance metrics: a comprison and validation, *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 4–13.

[73] **Tallent, N.R. and Mellor-Crummey, J.M.**, 2009. Effective performance measurement and analysis of multithreaded applications, *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, New York, NY, USA, pp. 229–240.

[74] **Frigo, M.**, **Leiserson, C.E. and Randall, K.H.**, 1998. The Implementation of the Cilk-5 Multithreaded Language, *In Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, pp. 212–223.

[75] **Fiutem, A.**, , **Antoniol, G.**, **Fiutem, R. and Cristoforetti, L.**, 1998. Using Metrics to Identify Design Patterns in Object-Oriented Software, *Proc. IEEE-CS Software Metrics Symp. (Metrics'98*, pp. 23–34.

[76] **Huston, B.**, 2001. The effects of design pattern application on metric scores, *Journal of Systems and Software*, **58**(3), 261–269.

[77] **Martin, R.**, 1994. OO Design Quality Metrics – An Analysis of Dependencies, *<http://www.objectmentor.com/resources/articles/oodmetrc.pdf>*, accessed at 11.04.2011.

[78] **Martin, R.C.**, 2002. *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall.

[79] **LEDA**. LEDA, *<http://www.algorithmic-solutions.com/leda/>*, accessed at 11.04.2011.

[80] **JBoss**. JBoss, *<http://www.jboss.org/>*, accessed at 11.04.2011.

[81] **DSpace**. DSpace, *<http://www.dspace.org/>*, accessed at 11.04.2011.

[82] **Mancoridis, S.**, **Mitchell, B.S. and Rorres, C.**, 1998. Using automatic clustering to produce high-level system organizations of source code, *In Proc. 6th Intl. Workshop on Program Comprehension*, pp. 45–53.

[83] **Xanthos, S.**, 2004. Identification of reusable components within an object-oriented software system using algebraic graph theory, *Proceedings of OPSLA '04*, ACM, New York, NY, USA, pp. 322–323.

[84] **Dietrich, J.**, **Yakovlev, V.**, **McCartin, C.**, **Jenson, G. and Duchrow, M.**, 2008. Cluster analysis of Java dependency graphs, *SoftVis '08: Proceedings of the 4th ACM symposium on Software visuallization*, ACM, New York, NY, USA, pp. 91–94.

[85] **Mitchell, B.S. and Mancoridis, S.**, 2006. On the automatic modularization of software systems using the Bunch tool, *Software Engineering, IEEE Transactions on*, **32**(3), 193–208.

[86] **Xiao, C. and Tzerpos, V.**, 2005. Software Clustering Based on Dynamic Dependencies, *Proceedings of CSMR '05*, IEEE Computer Society, Washington, DC, USA, pp. 124–133.

[87] **Wu, F. and Yi., T.**, 2004. Dependence analysis for UML class diagrams, *Journal of Electronics*, **21**, 249–254.

[88] **OMG**, 2007. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2, *<http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF>*, accessed at 11.04.2011.

[89] **Dhillon, I.S.**, **Guan, Y. and Kulis, B.**, 2007. Weighted Graph Cuts without Eigenvectors A Multilevel Approach, *IEEE Trans. Pattern Anal. Mach. Intell.*, **29**(11), 1944–1957.

[90] **Karypis, G.**, 2003. CLUTO - A Clustering Toolkit, *Technical Report* **#02-017**, University of Minnesota, Department of Computer Science.

[91] **Data Mining Laboratory, T.U.o.T.a.A.**, 2009. Graclus, <*http://www.cs.utexas.edu/users/dml/Software/graclus.html*>, accessed at 11.04.2011.

[92] **Karatzoglou, A.**, **Smola, A.**, **Hornik, K. and Zeileis, A.**, 2004. kernlab - An S4 Package for Kernel Methods in R, *Journal of Statistical Software*, **11**(9), 1–20.

[93] **van Dongen, S.**, 2008. MCL, <*http://micans.org/mcl*>, accessed at 11.04.2011.

[94] **Hubert, L. and Arabie, P.**, 1985. Comparing Partitions, *Journal of Classification*, **2**, 193–218.

[95] **Bergenti, F. and Poggi, A.**, 2000. Improving UML Designs using Automatic Design Pattern Detection, *Proc. 12th International Conf. Software Eng. and Knowledge Eng. (SEKE '00)*, pp. 336–343.

[96] **Antoniol, G.**, **Casazza, G.**, **Penta, M.D. and Fiutem, R.**, 2001. Object-oriented design patterns recovery, *Journal of Systems and Software*, **59**(2), 181 – 196.

[97] **Heuzeroth, D.**, **Holl, T.**, **Högström, G. and Löwe, W.**, 2003. Automatic Design Pattern Detection, *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, IEEE Computer Society, Washington, DC, USA, pp. 94–.

[98] **Balanyi, Z. and Ferenc, R.**, 2003. Mining Design Patterns from C++ Source Code, *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, pp. 305–314.

[99] **Tsantalis, N.**, **Chatzigeorgiou, A.**, **Stephanides, G. and Halkidis, S.T.**, 2006. Design Pattern Detection Using Similarity Scoring, *Software Engineering, IEEE Transactions on*, **32**(11), 896 –909.

[100] **Dong, J.**, **Zhao, Y. and Sun, Y.**, 2009. A Matrix-Based Approach to Recovering Design Patterns, *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, **39**(6), 1271–1282.

[101] **Nichols, B.**, **Buttlar, D. and Farrell, J.P.**, 1998. *Pthreads Programming*, O'Reilly.

[102] **GNU**. GNU C library's section on Limiting execution to certain CPUs, <*http://www.gnu.org/software/libc/manual/html_mono/libc.html#CPU-Affinity*>, accessed at 11.04.2011.

APPENDICES

APPENDIX A: Class Diagrams of Case Studies for Dependency Patterns
APPENDIX B: Graphs Extracted from Case Studies for Dependency Patterns

APPENDIX A

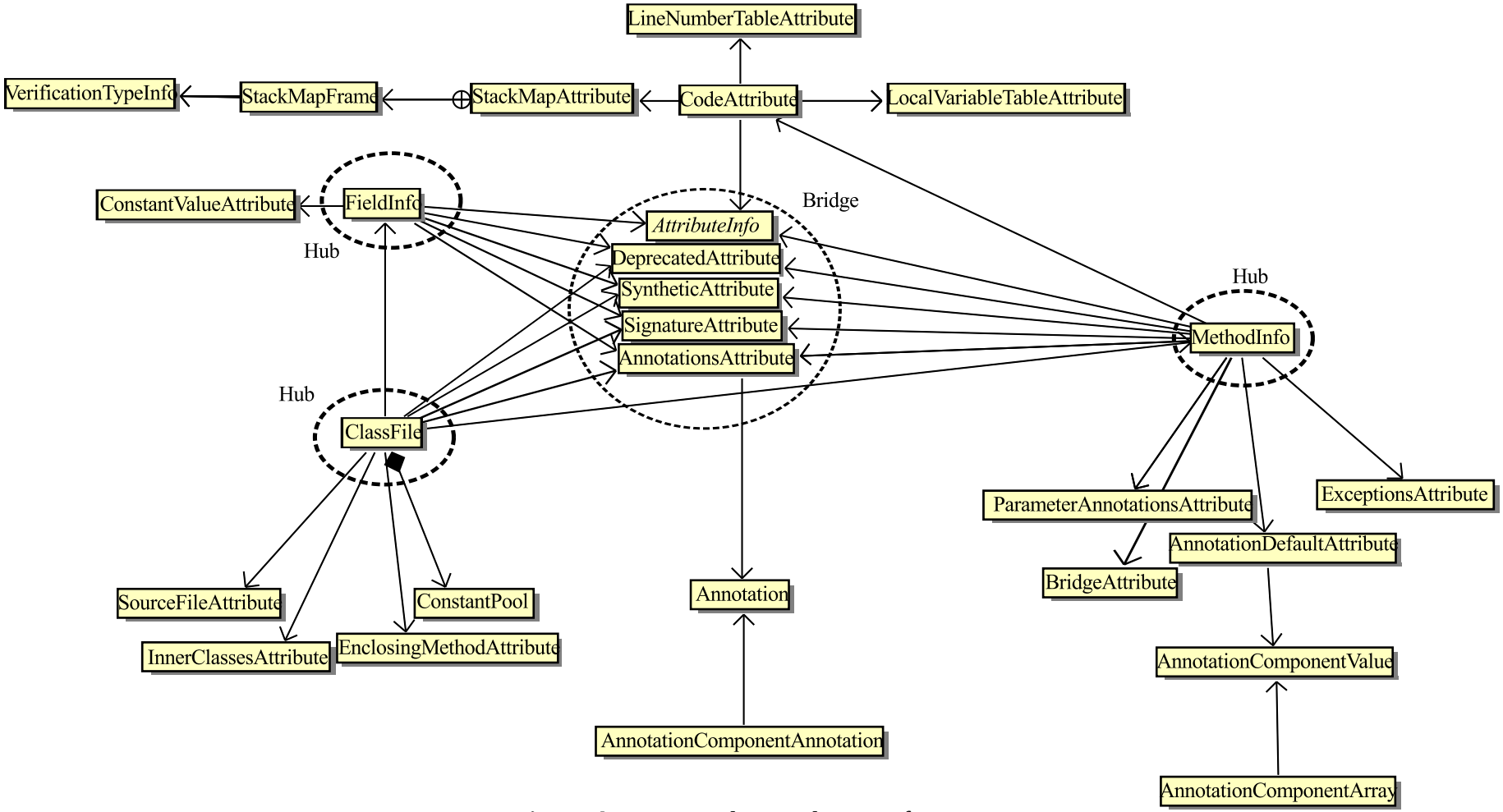


Figure A.1: Dependency relations of `CLASS`.

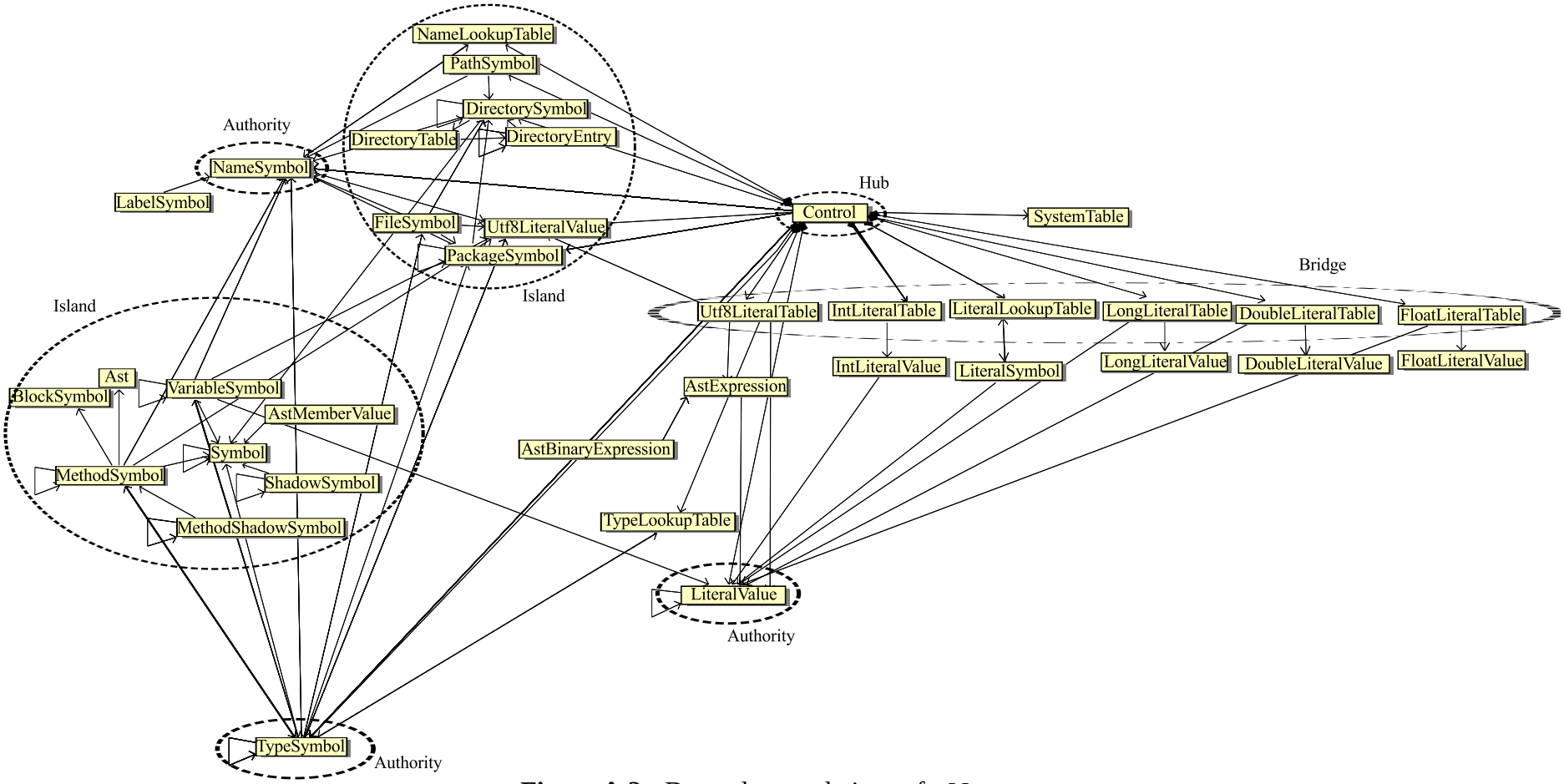


Figure A.2: Dependency relations of `LOOKUP`.

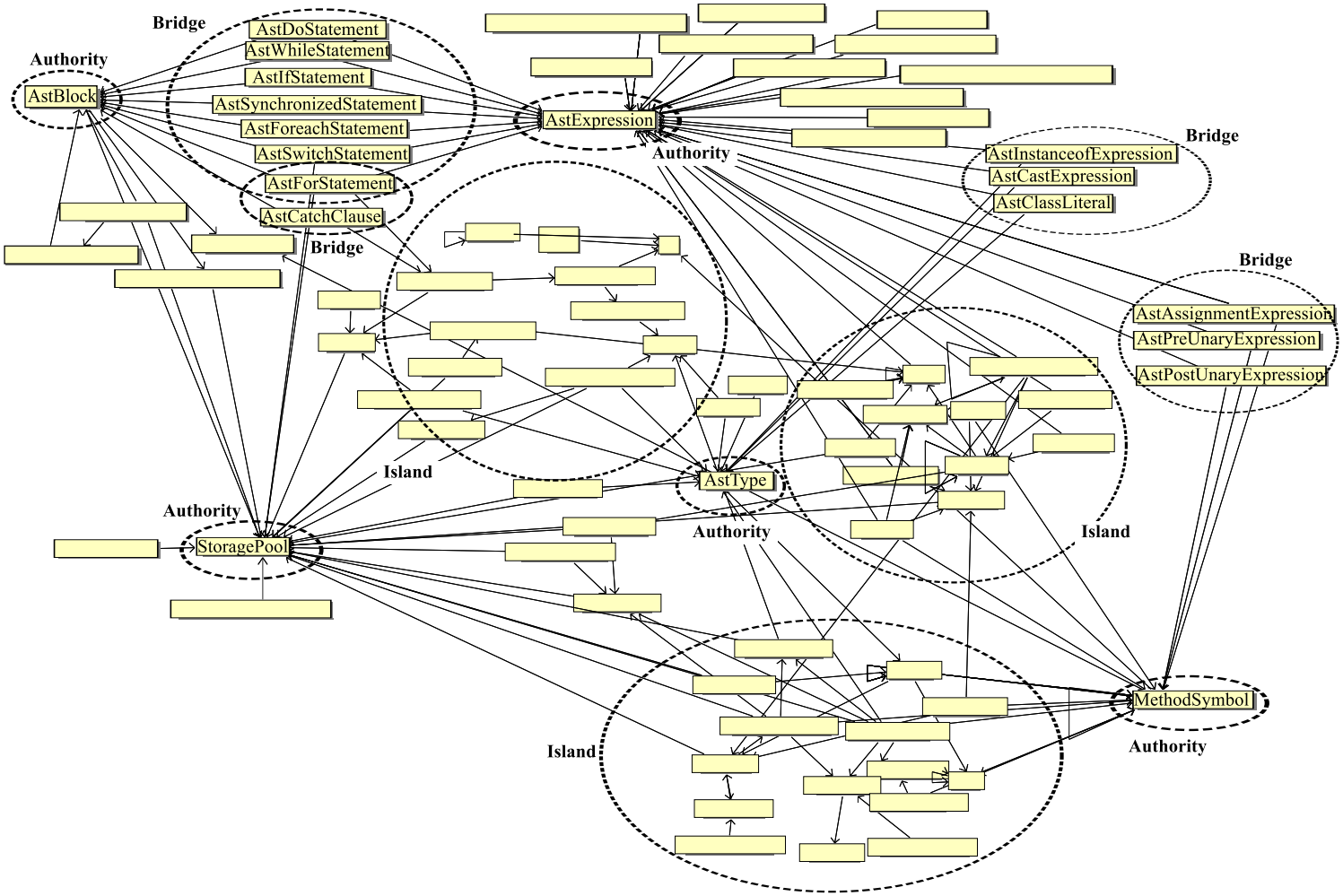


Figure A.3: Dependency relations of `AST` (Some insignificant class names have been excluded from the diagram for the sake of simplicity).

APPENDIX B

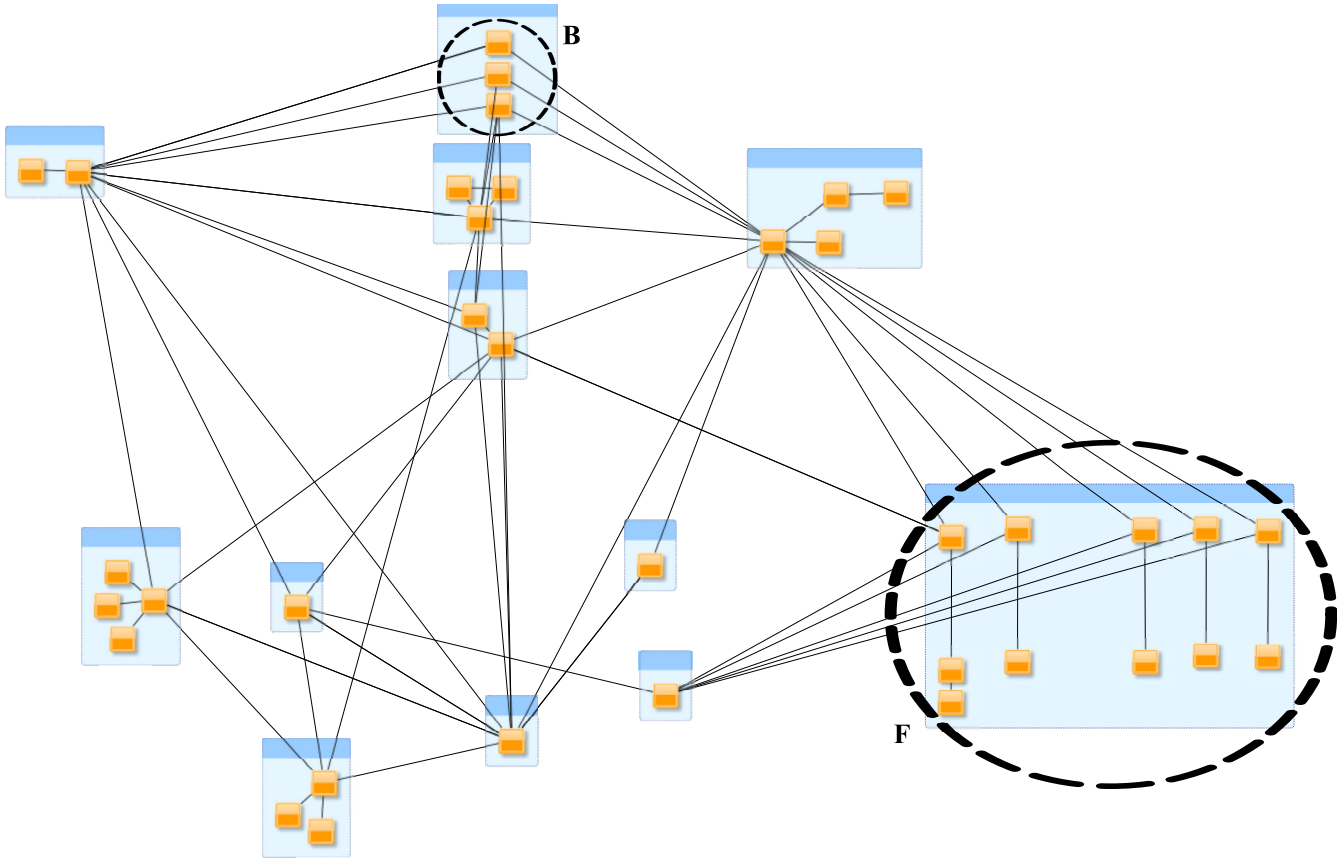


Figure B.1: Results of manual clustering for `LOOKUP`.

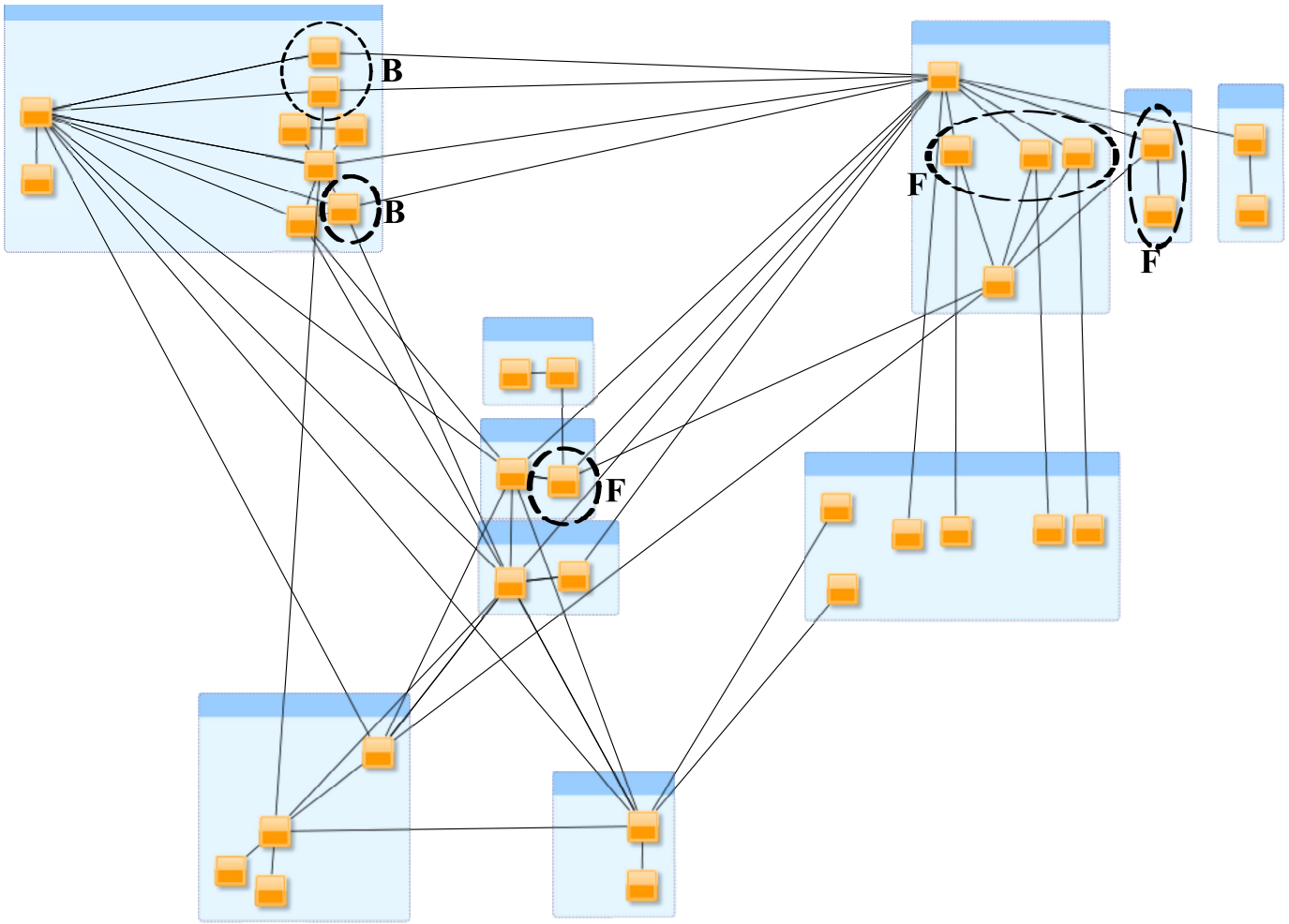


Figure B.2: Results of best clustering obtained for `LOOKUP`.

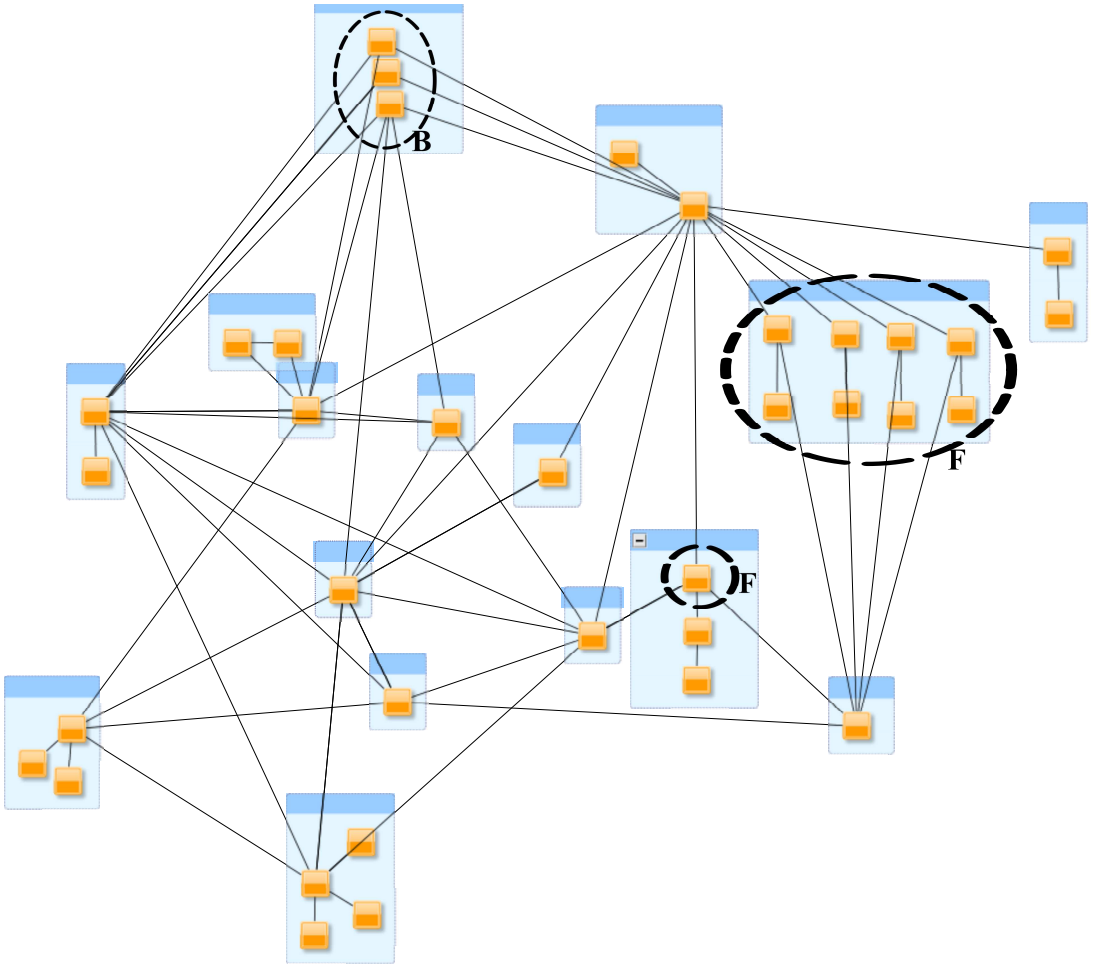


Figure B.3: Results improved by bridge detection for `LOOKUP`.

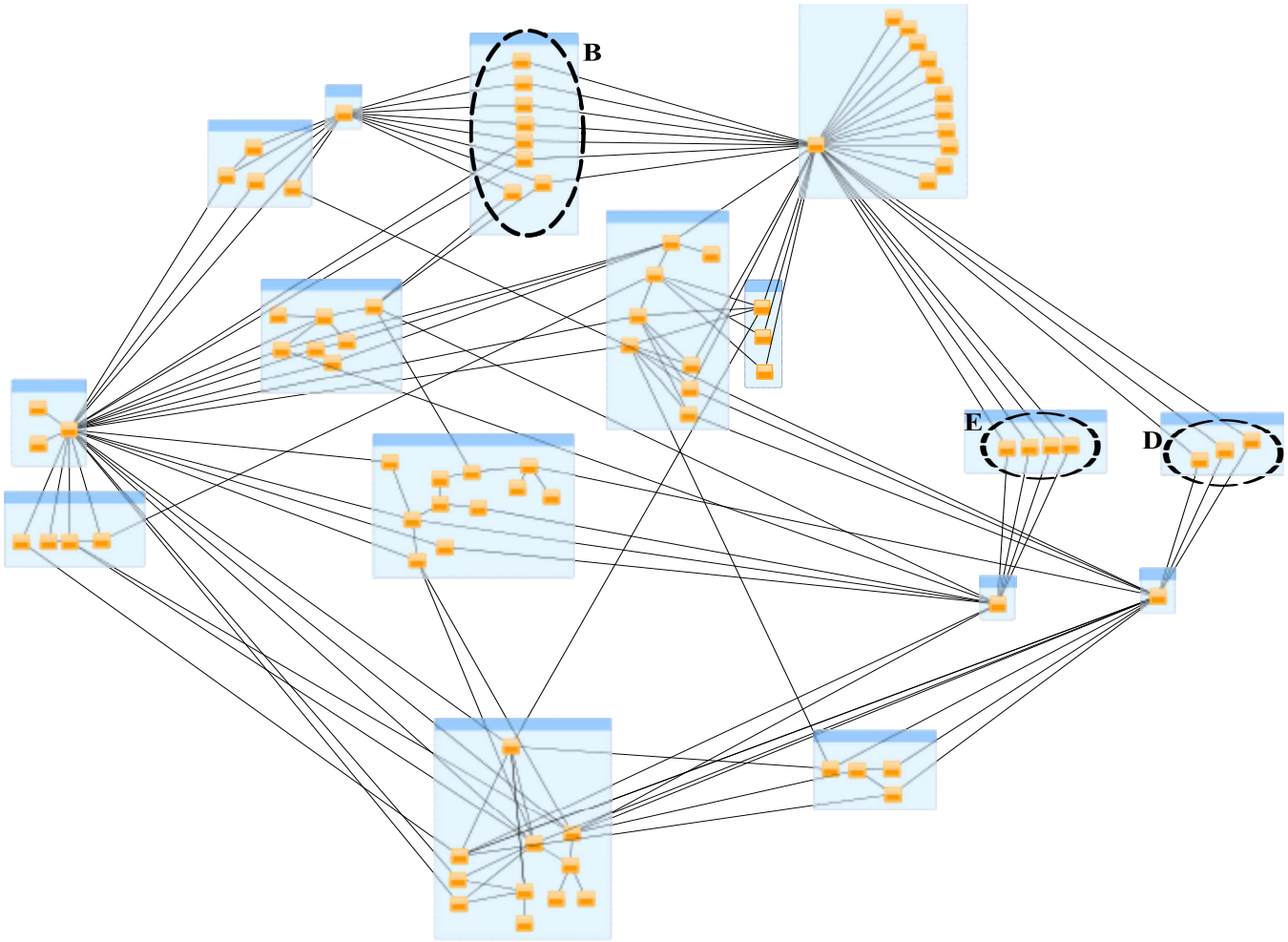


Figure B.4: Results of manual clustering for `AST`.

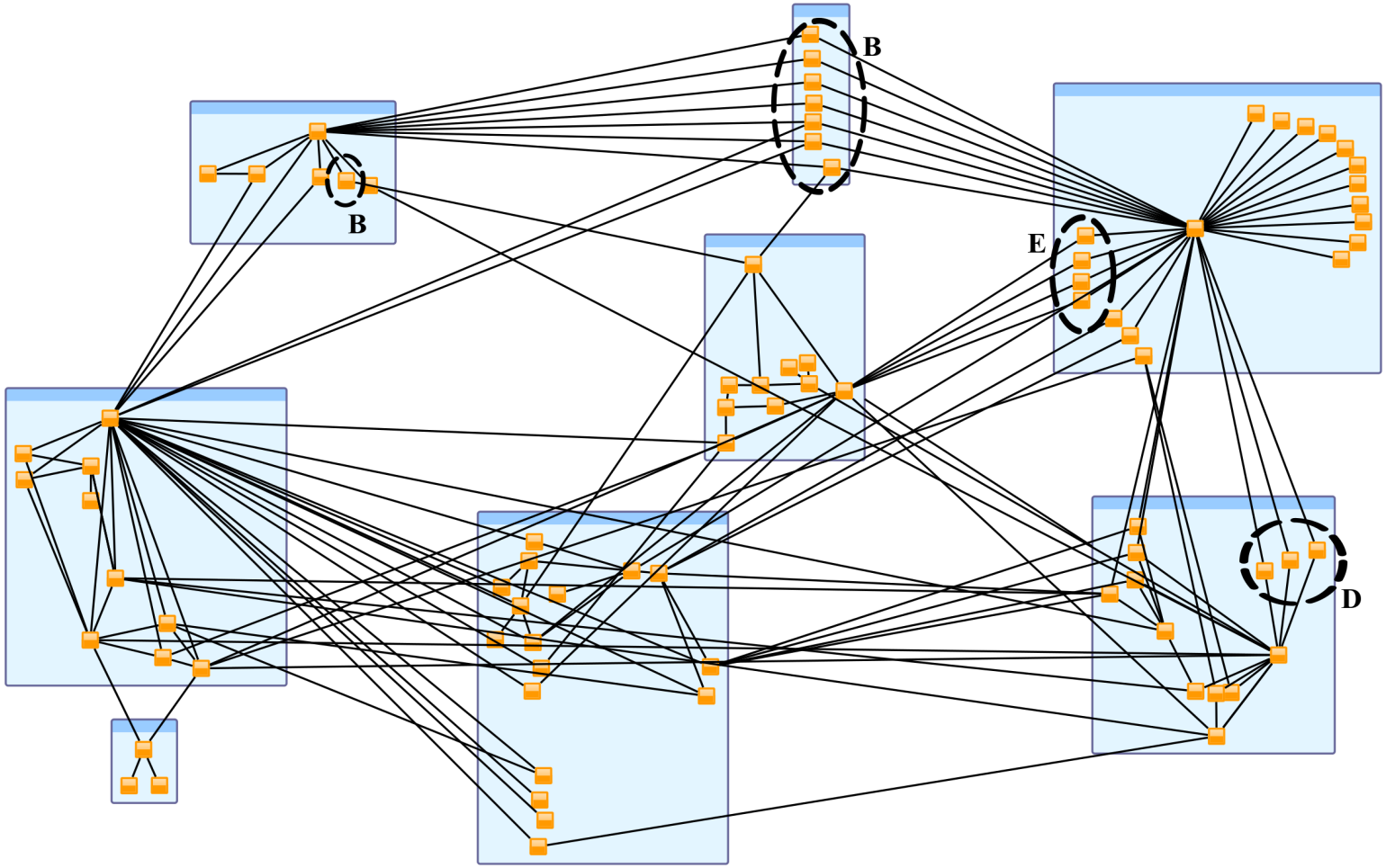


Figure B.5: Results of best clustering obtained for `AST`.

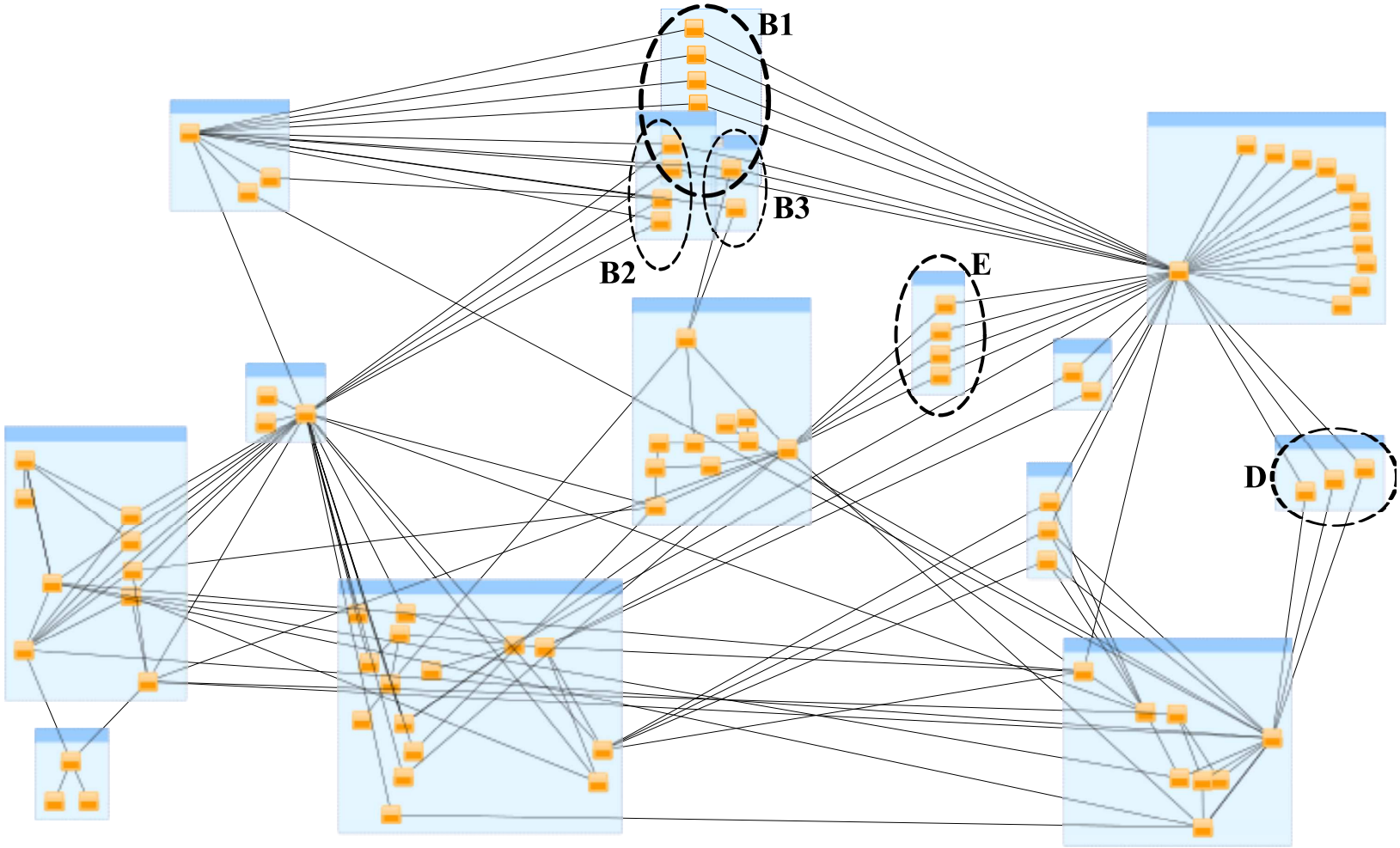


Figure B.6: Results improved by bridge detection for `AST`.

CURRICULUM VITAE

Candidate's full name: Tolga OVATMAN

Place and date of birth: Bursa, 13 June 1981

Universities and Colleges attended: Istanbul Technical University
M.Sc. in Department of
of Computer Engineering
(2003-2005,TURKEY)

Hacettepe University
B.Sc. in Department of
Computer Science and Engineering
(1999-2003,TURKEY)

Publications:

Tolga Ovatman, Thomas Weigert, Feza Buzluca, 2011: Exploring implicit parallelism in class diagrams: *Journal of Systems and Software*, Volume **84**, Issue 5, Pages 821-834, ISSN 0164-1212, DOI: 10.1016/j.jss.2011.01.005.

Tolga Ovatman, Aske W. Brekling, Michael R. Hansen, 2010: Cost Analysis for Embedded Systems: Experiments with Priced Timed Automata, *Electronic Notes in Theoretical Computer Science*, Volume **238**, Issue 6, Pages 81-95.

Tolga Ovatman, Feza Buzluca, 2011: "Model Driven Cache-Aware Scheduling of Object Oriented Software for Chip Multiprocessors", 08/2011, *14th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, Oulu, Finland.

Tolga Ovatman, Thomas Weigert, Feza Buzluca, 2010: "Applying Enhanced Graph Clustering to Software Dependency Analysis", 06/2010, *19th International Conference on Software Engineering and Data Engineering*, San Francisco, CA, USA, Received Software Engineering Track Best Paper Award.

Tolga Ovatman, Feza Buzluca,2009: "Software Design Pattern Behavior in Shared Memory Multiprocessor Systems", *International Conference on Computational Intelligence and Software Engineering*. dec/2009, s. 1–4.

Tolga Ovatman, Feza Buzluca, 2008: "Investigating software design pattern behavior in multiprocessor systems: A case study on observer", *23rd International Symposium on Computer and Information Sciences*. oct/2008, s. 1–4.

Tolga Ovatman, Feza Buzluca, 2008: "Çok Çekirdekli Sistemlerin Yazılım Kalitesi Üzerine Etkileri", *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu*, İstanbul.