Improving Efficiency, Scalability and Efficacy of Adaptive Computation Offloading in Pervasive Computing Environments

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy (Computer Science)

Ermyas Teshome Abebe B.App.Sc. (IT)

School of Computer Science and Information Technology,
College of Science, Engineering and Health
RMIT University
June 2012
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Ermyas Abebe
School of Computer Science and Information Technology
RMIT University
13 June 2012
Credits

Portions of the material in this thesis have previously appeared in the following publications:


Acknowledgements

So many people have helped or supported me along my candidature that neither the length of this page, nor that of this manuscript, would suffice to list them all. Nevertheless, I have attempted to list below those without whom this thesis would not be possible. Specifically I would like to thank:

- My senior supervisor, Dr Caspar Ryan, for his unlimited support throughout my candidature. I am especially appreciative of his valuable advice, which encompassed many aspects of my research, implementation, and writing.

- My second supervisor Dr Mikhail Perepletchikov who has offered me support of various forms towards the successful completion of my candidature.

- Professor Zahir Tari, who has served as a mentor, providing encouragement and advice as well as financial support for conference travel.

- My Parents, Teshome Abebe and Senait Abebe, for giving me everything a person could wish for; love, unending support and an excellent education.

- My fiancée Fiker Bayu, for her love, support, and unwavering patience through difficult times in my candidature.

- My friends and colleagues, especially Yohannes Yilma for his support and advice, Siyan Faisal and various members of the Distributed Systems and Networking Discipline for their inspiring research work.

- My uncle, Markos Abebe, for recognizing my passion for technology and buying me my first ever computer.

- RMIT University, for providing me with the financial means to undertake this degree.

- Hendrik Gani, and Pablo Rossi, for their work on Adaptive Computation Offloading, while at RMIT University. Their contributions have been the basis of the work presented in this thesis.

But perhaps most of all, I would like to thank God, for making all of this possible.
Contents

DECLARATION ........................................................................................................... II
CREDITS .................................................................................................................. III
ACKNOWLEDGEMENTS ........................................................................................ IV
LIST OF FIGURES .................................................................................................. IX
LIST OF TABLES .................................................................................................... XI
ABSTRACT .............................................................................................................. 1

CHAPTER 1 INTRODUCTION AND RATIONALE ........................................ 1

1.1 Example Scenario ............................................................................................... 3
1.2 Research Challenges in Adaptive Computation Offloading .................. 3
1.3 Scope and Focus of this Thesis ........................................................................ 5
  1.3.1 Sub-processes of Adaptation Decision Computation .................... 5
  1.3.2 Research Questions .................................................................................. 7
  1.3.3 Contributions ......................................................................................... 9

CHAPTER 2 LITERATURE REVIEW .............................................................. 12

2.1 Application Adaptation .................................................................................... 12
  2.1.1 Context and Context-Awareness ......................................................... 13
  2.1.2 Adaptation Decision Computation .................................................... 16
  2.1.3 Adaptation Variations .......................................................................... 17
2.2 Computation Mobility ...................................................................................... 21
  2.2.1 Computational Units of Mobility .......................................................... 23
2.3 Adaptive Computation Offloading ................................................................. 27
  2.3.1 Context Management for Adaptive Computation Offloading ...... 27
  2.3.2 Adaptation Decision Computation .................................................... 30
2.3.3 Adaptive Object Topology Computation .................................. 35

CHAPTER 3 IMPROVING EFFICIENCY AND SCALABILITY OF
ADAPTIVE CANDIDATE DEVICE SELECTION .......................... 41

3.1 RELATED WORK ........................................................................ 42
  3.1.1 One-way Adaptation (Surrogate based Adaptation) .............. 42
  3.1.2 Multi-way Adaptation (Peer-to-Peer Adaptation) ............... 42
3.2 LIMITATIONS OF EXISTING APPROACHES ......................... 46
3.3 DISTRIBUTED ADAPTIVE CANDIDATE DEVICE SELECTION .... 48
  3.3.1 Fitness Score Computation .................................................. 52
  3.3.2 Analysis of the proposed approach .................................... 55
3.4 EVALUATION ........................................................................... 57
  3.4.1 Environmental Settings ....................................................... 57
  3.4.2 Network Utilization ............................................................. 60
  3.4.3 Memory Utilization ............................................................. 63
  3.4.4 Power Utilization ............................................................... 64
3.5 SUMMARY ............................................................................... 66

CHAPTER 4 IMPROVING OBJECT TOPOLOGY COMPUTATION
EFFICACY ...................................................................................... 68

4.1 MOTIVATION ........................................................................... 69
4.2 EXISTING APPROACHES ........................................................ 72
  4.2.1 Object-level Granularity ...................................................... 72
  4.2.2 Class-level Granularity ....................................................... 73
4.3 PRELIMINARY INVESTIGATIONS ................................................. 78
4.4 HYBRID GRANULARITY GRAPHS ............................................. 80
  4.4.1 Object Graph Compaction ................................................. 81
5.4 Experiments and Results .................................................. 125
  5.4.1 Evaluation Materials and Scenario .............................. 125
  5.4.2 Results ...................................................................... 128
  5.4.3 Adaptation Decision Efficacy .................................... 136
5.5 Conclusion ..................................................................... 138

CHAPTER 6 Consolidated Adaptation Engine ..................... 139
  6.1 Consolidated Adaptation Engine (CAE) ......................... 140
    6.1.1 Fitness Score Computation .................................... 141
    6.1.2 Dynamic Hybrid Granularity Graph Decomposition .... 142
  6.2 Evaluation ..................................................................... 143
    6.2.1 Evaluation Materials .............................................. 144
  6.3 Results .......................................................................... 149
    6.3.1 Memory Utilization ................................................ 149
    6.3.2 Network Utilization ................................................. 151
    6.3.3 Power Utilization .................................................... 153
    6.3.4 Performance .......................................................... 156
    6.3.5 Efficacy ................................................................... 158
  6.4 Summary ................................................................. 159

CHAPTER 7 Summary and Conclusion ................................. 161
  7.1 Summary ................................................................. 161
  7.2 Practical Applicability .................................................. 163
  7.3 Future Work and Limitations ....................................... 164
List Of Figures

Figure 2-1: Local Adaptation Algorithm, Rossi and Ryan (Rossi P. and Ryan C., 2005) ................................................................................................................................. 36
Figure 2-2 Class Graph being partitioning on to three devices ................................. 39
Figure 3-1 Local Adaptation Approach ...................................................................... 44
Figure 3-2 Local Adaptation Algorithm basic flow, Rossi and Ryan (Rossi P. and Ryan C., 2005) ........................................................................................................... 45
Figure 3-3 Distributed adaptation decision-making algorithm ................................... 48
Figure 3-4 Distributed Candidate Selections and Adaptation Algorithm .................... 49
Figure 3-5 The environment scenario for which the existing approach provides better network utilization. nuexisting < nu(distributed) ........................................... 61
Figure 3-6 The environment scenario for which the distributed approach provides better network utilization. nudistributed < nu(existing) ................................. 61
Figure 3-7 Effectiveness of delay scheme vs degree to which existing outperforms distributed ........................................................... 63
Figure 3-8 Memory overhead saved by the distributed algorithm compared to the existing algorithm ................................................................. 64
Figure 3-9 Power utilization comparison of proposed algorithm vs. existing algorithm ........................................................................................................... 66
Figure 4-1 A class diagram of a simple application consisting of three classes ...... 74
Figure 4-2 Class Graph partitioned to three devices ................................................. 75
Figure 4-3 Barnes-Hut implementation class graph |V|=8, |E|=15 .............................. 79
Figure 4-4 Barnes-Hut implementation object graph |V| = 2408, |E|=3604 ............... 80
Figure 4-5 Granularity Decomposition Diagram ...................................................... 81
Figure 4-6 Example Class Graph showing criteria 1-3 from the perspective of a heavy class X ................................................................................................. 83
Figure 4-7 Hybrid Graph Granularity generation algorithm ..................................... 88
Figure 4-8 Class Graph consisting of three vertices with a single heavy vertex... 89
Figure 4-9 Fissuring of class 1 (from 0) into class 1’ and class 1'' .......................... 89
Figure 4-10 NASA WWJ Class Graph ....................................................................... 90
Figure 4-11 NASA WWJ Hybrid Granularity Graph. Fissured vertices are shaded black ............................................................................................................. 90
Figure 4-12 Memory utilization of the application graphs maintained ...................... 96
Figure 4-13 Object Topology Computation Performance of Class, Object and Hybrid Granularity Graphs. HGG performance includes decomposition time,...................................................................................................................... 97
Figure 4-14 Object Topology Computation Power Utilization result ...................... 98
Figure 4-15 Edge-Cut results for evaluated applications ........................................ 100

IX
Figure 5-23 Edge-Cut Comparison of Non-Distributed (Existing) vs Distributed (Proposed) Approaches, During Adaptation of NASA World Wind Java Application

Figure 6-1 Collaboration-Wide Memory Utilization of EAE, CAE-A and CAE-B

Figure 6-2 Network Utilization Comparisons of EAE versus CAE

Figure 6-3 Power utilization on adapting device (HTC Dream) resulting from factor EAE 1) and CAE 1)

Figure 6-4 Power utilization on candidate device (HTC HD2) resulting from factor EAE 1) and CAE-1)

Figure 6-5 Decision Computation and Graph Decomposition Power Usage (A1, A2)

Figure 6-6 Adaptation Decision Computation and Graph Decomposition Power Utilization (A3-A6)

Figure 6-7 Total Power Utilization on HTC G1

Figure 6-8 Performance Comparisons for Applications A1 and A2

Figure 6-9 Performance Comparisons for Applications A3-A6

Figure 6-10 Efficacy Results of CAE-A, CAE-B and EAE

List Of Tables

Table 2-1 Environmental Resource Usage Metrics Proposed by (Rossi and Tari, 2007)

Table 3-1: Variables Influencing Network Utilization and Memory Utilization in Both Algorithms

Table 3-2 The Effect of a Delay Scheme on the Network Utilization Disparity Between the Algorithms

Table 4-1 Software Metrics of Objects $O_1$, $O_2$ and $O_3$

Table 4-2 Coupling Amongst Objects $O_1$, $O_2$ and $O_3$

Table 4-3 Alternative Configurations for Simple Applications Between Two Devices

Table 4-4 Selected Corpus of Applications for Evaluation. Applications from the J-Olden Benchmark Suite and NASA WWJ

Table 5-1 Application Statistics in Evaluation Scenario

Table 6-1 Randomly Generated Application Graphs

Table 6-2 Generated Resource Utilization Specifications for Application Graphs

Table 6-3 Application Granularity Generation Thresholds

Table 6-4 Generated Hybrid Granularity Application Graphs

XII
Abstract

As computing becomes more mobile and pervasive, there is a growing demand for increasingly rich, and therefore more computationally heavy, applications to run in mobile spaces. However, there exists a disparity between mobile platforms and the desktop environments upon which computationally heavy applications have traditionally run, which is likely to persist as both domains evolve at a competing pace. Consequently, an active research area is Adaptive Computation Offloading or cyber foraging that dynamically distributes application functionality to available peer devices according to resource availability and application behaviour.

Integral to any offloading strategy is an adaptive decision making algorithm that computes the optimal placement of application components to remote devices based on changing environmental context. As this decision is typically computed by constrained devices and may occur frequently in dynamic environments, such algorithms should be both resource efficient and yield efficacious adaptation results. However, existing adaptive offloading approaches incur a number of overheads, which limit their applicability in mobile and pervasive spaces.

This thesis is concerned with improving upon these limitations by specifically focusing on the efficiency, scalability and efficacy aspects of two major sub processes of adaptation: 1) Adaptive Candidate Device Selection and 2) Adaptive Object Topology Computation. To this end, three novel approaches are proposed.

Firstly, a distributed approach to candidate device selection, which reduces the need to communicate collaboration metrics, and allows for the partial distribution of adaptation decision-making, is proposed. The approach is shown to reduce network consumption by over 90% and power consumption by as much as 96%, while maintaining linear memory complexity in contrast to the quadratic complexity of an existing approach. Hence, the approach presents a more efficient and scalable alternative for candidate device selection in mobile and pervasive environments.

Secondly, with regards to the efficacy of adaptive object topology computation, a new type of adaptation granularity that combines the efficacy of fine-grained adaptation with the efficiency of coarse level approaches is proposed. The approach
is shown to improve the efficacy of adaptation decisions by reducing network overheads by a minimum of 17% to as much 99%, while maintaining comparable decision making efficiency to coarse level adaptation.

Thirdly, with regards to efficiency and scalability of object topology computation, a novel distributed approach to computing adaptation decisions is proposed, in which each device maintains a distributed local application sub-graph, consisting only of components in its own memory space. The approach is shown to reduce network cost by 100%, collaboration-wide memory cost by between 37% and 50%, battery usage by between 63% and 93%, and adaptation time by between 19% and 98%.

Lastly, since improving the utility of adaptation in mobile and pervasive environments requires the simultaneous improvement of its sub processes, an adaptation engine, which consolidates the individual approaches presented above, is proposed. The consolidated adaptation engine is shown to improve the overall efficiency, scalability and efficacy of adaptation under a varying range of environmental conditions, which simulate dynamic and heterogeneous mobile environments.
Chapter 1  Introduction and Rationale

The accelerated growth and cohesive integration of fast mobile computing hardware, high-speed wireless internet infrastructure and open mobile operating platforms, have brought about the expedited mainstream adoption of smart mobile computing, and the consequent proliferation of mobile devices which are poised to exceed PC sales by 2012 (Mary Meeker et al.). In concert, these trends signal a fundamental shift of personal computing away from fixed desktop environments and into mobile and pervasive computing spaces.

Subsequently, there is increasing expectation that these devices will offer the same comprehensive and self-contained computing experiences offered by current desktop computing software. Partially enabling this requisite are mobile applications, which have over the years evolved from trivial accessories in old feature phones to indispensable tools for smart computing. Today, these applications offer users a wide spectrum of capabilities from entertainment and social networking to simple document and image processing; often leveraging emergent sensor capabilities (GPS (Hofmann-Wellenhof et al., 1997; Kaplan and Hegarty, 2006), Accelerometers (Khan et al., 2010), and Near Field Communication (Michahelles et al., 2007; Ortiz Jr, 2006) etc.) to deliver increased social and mobile computing experiences (Khan et al., 2010; Strommer et al., 2006; Wagner and Schmalstieg, 2009a, b).

Nevertheless, enabling a seamless transition to a post-desktop model of computing requires mobile applications to offer comparable software capabilities to desktop applications. This is however challenging as such applications require the processing power of desktop devices, which are orders of magnitude more powerful than their mobile counterparts. Moreover, this mismatch is likely to persist as PCs advance at a pace affirming Moore’s law (Powell, 2008) whilst mobile hardware advancements are curbed by relatively slow progress in battery life extension research (Chareen et al., 2008).

Hence, an approach that allows mobile devices to transcend hardware limitations in order to execute complex and heavy applications is desirable. An ostensible approach to this problem is a thin-client mode of operation (Jing et al., 1999), in which the core functionalities of an application are performed in the Cloud (Hayes, 2008), while a mobile device presents a minimal user interface through which these functionalities can be invoked and their outputs displayed. While this can increase computational flexibility, it has various limitations. Firstly, extensive development cost is incurred in reengineering existing standalone applications for a client-server
mode of execution; which, in some cases, impedes or retards the transition into mobile spaces. Secondly, the use of a dedicated server presents a single point of failure, which is limiting in mobile spaces where frequent disconnections are likely as a result of user mobility. Lastly, a thin-client approach underutilizes the growing resource availability of mobile devices, which could otherwise be used to minimize the use of power hungry servers to yield greener computing alternatives (Swanson and Taylor, 2011).

Hence, this thesis focuses on a more sophisticated alternative, which overcomes these limitations by bringing together two distinct concepts, Application Adaptation (Bharghavan and Gupta, 1997; Kakousis et al., 2010; Redmond and Cahill, 2006) and Computation Mobility (Brooks, 2004; Dror Garti et al., 2000; Fuggetta et al., 1998; Garti et al., 2000; Herbert et al., 1999). Application Adaptation is the ability of an application to dynamically reconfigure its behaviour in response to changes in an execution environment. For instance, an application might reduce its network consumption in a bandwidth scarce environment (Kim and Copeland, 2003), or spread out its memory usage across multiple devices in the event that a device runs out of memory (Hütter and Moschny, 2008; Michael Philippsen and Haumacher, 1998; Xiaohui Gu et al., 2004). Computation Mobility (Fuggetta et al., 1998) on the other hand, is a strategy in which a device executes a given application by first distributing subsets of its runtime objects to other devices (performed either offline or manually at runtime). This can be done for a number of reasons including application behaviour extension, load mitigation, and performance improvement, whereby in each case the device extends its computational capabilities by utilizing externally available resources.

Hence, in Adaptive Computation Mobility (also known as Adaptive Computation Offloading or Cyber foraging) (Gu et al., 2003; Li et al., 2001; Ou et al., 2007a; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) a distribution of components to devices is performed and constantly calibrated, in order to maximize context specific optimizations. This adaptive re-configuration of component distributions is imperative in resource dynamic environments (such as mobile spaces) where frequent changes in resource availability could quickly render previous component distributions suboptimal and costly.

Before proceeding further into Adaptive Computation Offloading, a simple real world scenario is provided in section 1.1 below, so as to illustrate the rationale behind the approach and present an example through which the spectrum of current research
challenges (section 1.2) can be outlined and the specific focus of this thesis identified (section 1.3).

1.1 Example Scenario

Donald is playing an immersive game when he realizes he needs to leave for work. Being an avid gamer, Donald decides he wants to continue gaming as he commutes to the office, which is about an hour’s train ride away. He pauses his game and starts an adaptive offloading middleware, which automatically transfers his game (and its state) to his tablet PC. However, having determined that Donald’s tablet is too constrained for the game, the offloading middleware forages for available external devices and then distributes portions of the game without compromising its fidelity (user experience and application performance).

As Donald walks to the train station, the connectivity between his device and external collaborating devices constantly changes. For instance, as he leaves his apartment, his device switches from his home Wi-Fi to a relatively slower 3G network. Similarly, as he performs other operations on his tablet (Email checking, browsing etc.) the resource availability on his device diminishes. Meanwhile however, a lightweight offloading middleware running on Donald’s tablet detects these changes and ensures that the distribution of components between Donald’s tablet and the remote devices constantly evolves to ensure optimal application performance despite these changes.

Upon arriving at the train station, Donald resumes his game, and soon after, enters his departing train. In the background, the middleware detects reduced bandwidth over 3G as Donald’s train enters a subway, and in response switches to Ad-Hoc wireless connectivity (or Bluetooth) and distributes the game’s components to other mobile devices within the train. As Donald advances from one game level to another, the underlying middleware optimizes the distribution of the game across the collaborating devices, in order to ensure a maximal game playing experience.

1.2 Research Challenges in Adaptive Computation Offloading

This section uses the above example to outline some of the research challenges in Adaptive Computation Offloading in a real-world context, so as to later aid in identifying the specific focus of this thesis (section 1.3).
Development Transparency: In the above scenario, Donald’s initial transfer of the game to his Tablet PC was a seamless process, which did not require the development or installation of separate software on his tablet. This ability to automatically transform traditional non-distributed software into an adaptive and distributed version of itself, thereby removing associated development effort, is termed as Development Transparency (Gani and Ryan, 2009; Geoffray et al., 2006; Ryan C. and Westhorpe C., 2004). This capability would facilitate a cost effective transition of existing desktop software into mobile spaces, with little or no modification. The major challenges in such an approach involve the degree of transparency achievable as measured through the development effort required, the efficiency of the transformation process and the reduction of the extra code introduced in the transformed application.

Context Information Collection (Metrics Collection): Adaptation requires an awareness of changes in the environment so as to react accordingly when necessary. In the case of Adaptive Computation Offloading, this is achieved through the measurement and exchange of context information (Hong et al., 2009) such as the resource availability on devices and their underlying network infrastructure (Environmental Metrics (Gani H. et al., 2006)), as well as the resource requirements of individual application components (Software Metrics (Gani H. et al., 2006)). While frequently measuring this information is essential to account for resource dynamism in mobile spaces, it incurs notable overheads. In addition, an accurate representation of context information to allow for predictability of future changes is also essential. Hence recent works have focused on the efficiency, accuracy and predictability aspects of context management (Gani H. et al., 2006; Rossi and Tari, 2007).

Security and Privacy: the placement of an application’s components (objects or classes) on foreign devices raises privacy and security concerns; both from the perspective of securing the integrity of the application, as well as protecting remote devices from potentially malicious code (e.g. through Man-in-the-middle attacks). Some security techniques being explored include digital certificates, sandboxing and code obfuscation (Brooks, 2004; Necula, 1997).

Fault Tolerance and Availability: Frequent disconnection, which is partly a result of user mobility, is common in mobile spaces. Hence, adaptation solutions need to ensure that the disconnection of a single device does not result in the total failure of an application. A common approach to this problem is component replication (Katmon and Ryan, 2011; N. Narasimhan et al., 2000), in which copies of components are placed on different devices for redundancy. Some of the main
challenges in replication include the efficiency of the replication process, and the reduction of replica management overheads (synchronizing replica states etc.)

Adaptation Decision Computation: An adaptation decision is computed whenever changes in the environment render an existing component-to-device mapping suboptimal (E.g. A device runs out of resources). Efficiently computing such decisions is imperative because of the constraint of the devices computing them, and the frequency at which such decisions must be computed to account for user mobility and resource fluctuations. In addition, the efficacy of a computed decision, as measured through the resource cost of the component distributions and the performance of an adapting application, is also an essential factor, which determines the viability of Adaptive Computation Offloading in mobile spaces. However, the efficacy of adaptation decisions and the efficiency of the underlying process are often trade-offs. As a result, existing approaches incur overheads in one or both of these factors, which limit their overall utility in mobile spaces. This thesis focuses on this aspect of Adaptive Computation Offloading, and is hence discussed in more detail in the following sub-section.

1.3 Scope and Focus of this Thesis

From the breadth of research foci discussed above this thesis focuses on improving Adaptation Decision Computation by optimizing the efficiency and scalability of the process and improving the efficacy of its outcomes. As briefly discussed earlier, these factors are imperative for mobile devices in resource dynamic environments since adaptation decisions must be performed frequently (efficiency) to address resource changes and must often optimize for an unbound number of collaborating devices and application components (scalability and decision efficacy). Furthermore, any inefficiency in the process, or sub optimality in the computed decisions directly translates into power consumption, financial cost (bandwidth cost, external resource sharing cost etc.) and reduction in application performance.

1.3.1 Sub-processes of Adaptation Decision Computation

Improving the overall efficacy of adaptation decision computation requires optimizing each of its sub-processes (steps). Generally, a device computing an adaptation decision follows two steps: A) Selecting a set of candidate devices which can offer an optimal set of resources, and then B) Computing a component to device mapping (component topology), which satisfies one or more desired objectives. A
brief discussion of each sub-process is provided below so as to provide the requisite background, and identify the existing limitations which are to be addressed in this thesis.

A. Candidate Device Selection

When computing an adaptation decision, a client first selects a subset of devices from within the collaboration that can offer it the most optimal set of resources. In order to compute such a decision, a device needs to know the temporal context information of each external device (Environment Metrics), such as its power, memory, network and processor availabilities. This is achieved by getting each device to frequently measure (see Context Collection in section 1.2) and then communicate its environment metrics to the rest of the collaboration. When a device needs to adapt, it uses these metrics, to compute the relative optimality of each device in relation to its own adaptation objectives; and accordingly selects the most optimal set of candidate devices.

However, this process has various limitations: Firstly, the need to frequently communicate collaboration wide context information, and then store it on each device incurs efficiency overheads which rise as a factor of collaboration and context information sizes as well as the frequency of metrics communication (which is a result of resource fluctuation in the collaboration). Secondly, the process of determining the most optimal subset of devices is a CPU intensive process, which reduces the performance of the adaptation process. Lastly, computing an efficacious decision (as measured by the optimality of the selected devices relative to the rest of the collaboration) is a computationally intensive process, which must often be conceded to the efficiency and scalability of the process producing it, thus potentially leading to suboptimal adaptation outcomes. This thesis aims to address these limitations by focusing on the specific research questions identified in section 1.3.2.A.

B. Component Topology Computation (component to device mapping)

Once a set of suitable target devices has been selected, the second step involves determining an optimal distribution of components across collaborating devices. Component topologies are computed based on abstract representations of an application’s runtime objects and their behaviour (resource usage, coupling patterns etc.). This representation is often modelled as a graph in which vertices represent
components and edges represent their couplings; and is often kept updated on each device through a combination of component profiling (Software Metrics collection for local components hosted on a given device) and remote updates from external devices (for components hosted on other devices). The graph can either be fine-grained, whereby a vertex exists for each object or coarse-grained where a single vertex exists for a class and all its objects. When a device adapts, it uses this context information, along with information it possesses about external devices (see 1.3.1.A above) to compute an optimal component-to-device mapping.

An efficacious component topology both improves application performance and reduces the inter-device communication resulting from a component distribution (remote component invocation); and is thus essential for adaptation in mobile spaces.

However, various inefficiencies encumber this process. Firstly, maintaining and communicating component related context information incurs resource costs proportional to the size (number of components) and variability (in resource consumption over time) of an application. Secondly, existing topology computation heuristics are CPU and power intensive processes which compromise the efficacy of adaptation decisions for computational feasibility; thus yielding potentially suboptimal adaptation outcomes.

This thesis aims to address these limitations by focusing on the specific research questions identified in section 1.3.2.B.

1.3.2 Research Questions

This thesis aims to improve the overall utility of the adaptation decision computation process by first improving on each of its sub processes separately and later combining these optimization into an integrated adaptation decision computation engine. Hence, the specific research questions below, 1.3.2.A and 1.3.2.B, correspond to the two sub processes discussed in 1.3.1.A and 1.3.1.B above, whereas research question 1.3.1.C focuses on improving the overall adaptation decision computation process.

A. How can the candidate device selection process be improved?

The overall utility of the candidate device selection process is a factor of its efficiency, scalability and decision efficacy. As a methodical approach to the problem, these factors are addressed separately in Questions A.1 and A.2, and a means of combining these optimizations explored in Question C.
A.1. **How can the efficiency and scalability of the candidate device selection process be improved?**

This section investigates approaches to reducing the cost of metrics maintenance and communication while improving the performance of the candidate device selection process (*efficiency*). It also aims to improve the *scalability* of the process with regards to 1) The collaboration size 2) The application size 3) the size of device metrics (which corresponds to diversity of devices and the amount of information they must communicate as a result) and 4) the degree of resource fluctuation in a collaboration (higher resource fluctuation often requires more frequent context information communication).

A.2. **How can the efficacy of the candidate device selection process be improved?**

This section investigates approaches to improving the *efficacy* of the candidate devices selected under a number of environment conditions including 1) changing collaboration sizes 2) changing application sizes and 3) different size and number of environmental metrics considered.

B. **How can the component topology computation process be improved?**

The overall utility of a component topology decision is a factor of both the *efficiency* and *scalability* of the process, and the *efficacy* of its decisions. Similar to Question A, these aspects are addressed separately in Questions B.1 and B.2 and a means of combining these optimizations later explored in Question C.

B.1. **How can the efficiency and scalability of the component topology computation process be improved?**

This section aims to improve the efficiency of component topology computation by reducing the overheads associated with storing, maintaining and communicating the abstract representations required for topology computation, and improving the performance of decision heuristics. It also focuses on improving scalability with regards to increasing application (number of components) and collaboration sizes.

B.2. **How can the efficacy of the component topology computation process be improved?**
The granularity at which adaptation decisions are computed impact both the efficacy and the efficiency of the adaptation process. While fine granularity (object-level) adaptation offers improved efficacy, it incurs prohibitive computational costs. On the other hand, coarse granularity (class-level) offers efficient topology computation but reduces the efficacy of adaptation outcomes. Hence, this section focuses on leveraging the efficacy of fine-grained adaptation while minimizing the resultant computation overheads.

C. How can the efficiency, scalability and efficacy of Adaptive Computation Offloading decision computation be simultaneously improved?

As discussed in section 1.3.1, simultaneously improving the efficiency, scalability and decision efficacy of the candidate device selection or component topology computation sub-process, is an optimization challenge that must be achieved in order to improve the overall utility of adaptive offloading. This is because these factors are often trade-offs; and hence, any gains from improving one aspect (efficiency, or efficacy) could be negated by losses in another. Moreover, improvements in these aspects must be achieved for both sub-processes (Candidate Device Selection and Component Topology Computation) so as to improve the overall adaptation decision computation process.

Hence, this section focuses on 1) simultaneously improving efficiency, scalability and efficacy of candidate device selection 2) simultaneously improving the efficiency, scalability and efficacy of component topology computation and 3) Integrating the improvements in 1) and 2) into a unified adaptation engine which improves the utility of Adaptive Computation Offloading in mobile environments.

1.3.3 Contributions

To address the above research questions, three novel approaches are proposed in this thesis; and are discussed in sections 1.4.A and 1.4.B below, which correspond to research questions 1.3.2.A and 1.3.2.B respectively.

A. Improving candidate device selection process

A novel distributed approach to selecting candidate devices is proposed (Chapter 3), which reduces the need to communicate collaboration metrics (unlike existing
approaches), and allows for the partial distribution of the candidate device selection process. A comparative evaluation of this approach against existing work shows that it improves not only the efficiency, and scalability of the candidate selection process but also the efficacy of the candidate devices selected; thereby addressing questions A.1, and A.2 in section 1.3.2.

B. Improving component topology computation process

In order to address research Question B) in section 1.3.2, two novel approaches are proposed: one for improving the efficiency and scalability (B.1) of the process and another for improving its decision efficacy (B.2). These are later combined in C below, to improve the overall utility of the component topology computation process.

B.1. Improving the efficiency and scalability of component topology computation

A novel distributed approach for representing components and their runtime behaviour is proposed, where each device only maintains vertex representations for local components, while storing stub like abstractions (cloud vertices) for components on external devices. The approach removes the need to maintain a complete representation of an application as well as the need to communicate related updates to the collaboration. In addition, a localised decision computation heuristic is proposed which can utilize this new representation to compute efficacious adaptation decisions. An evaluation of the approach against existing work showed that it improves both efficiency and scalability of the component topology computation process, as well as, in some cases, the efficacy of adaptation decisions computed. While this directly addresses question B.1 under section 1.3.2 it also partly addresses B.2 by improving the efficacy of adaptation decisions computed under certain scenarios.

B.2. Improving the efficacy of component topology computation

To improve the efficacy of component topologies generated, a new level of granularity for representing an application’s state is proposed, which combines the efficacy of fine grained (object level) adaptation with the efficiency of coarse grained (class level) approaches. An approach for deriving this level of granularity through the dynamic decomposition of a runtime coarse-granularity representation is also proposed. An evaluation of the approach showed improvements in the efficacy of
adaptation decisions under all evaluated cases, as measured through the reduction in network overheads and the improvement of application performance. In terms of efficiency, while the approach performed orders of magnitude better than fine-grained (object-level) adaptation it incurred modest overheads in comparison to coarse-level approaches.

C. Simultaneously Improving efficiency, scalability and efficacy of adaptive offloading decision computation

By effectively integrating the candidate device selection approach proposed in (A) with the distributed component representation approach proposed in (B.1.) and the hybrid granularity strategy proposed in (B.2), a new Consolidated Adaptation Engine (CAE) is proposed (Chapter 6).

Through an evaluation consisting of a diverse range of applications adapting in a heterogeneous mobile collaboration, the approach is shown to simultaneously improve the efficiency, scalability and decision efficacy of the overall adaptation process as compared to an existing state-of-the-art approach. This contribution thus addresses Question C in section 1.3.2 and consequently satisfies the main objective of this thesis.
Chapter 2  Literature Review

As briefly discussed in Chapter 1, Adaptive Computation Offloading is a combination of two distinct concepts, Application Adaptation and Computation Offloading. This chapter first discusses the breadth of research foci of each of these two sub-processes in order to provide the requisite background for discussing Adaptive Computation Offloading approaches and the core limitations that this thesis aims to address.

Specifically, in section 2.1, Application Adaptation is discussed in terms of its diverse implementation in existing literature, while identifying common underlying sub-processes which impact software quality factors of importance to this thesis (efficiency, scalability and decision efficacy). This sets the rationale for adaptation in resource dynamic mobile environments and introduces the concepts of context information management and adaptation reasoning, which are later expounded in the context of Adaptive Computation Offloading strategies in section 2.3.

In section 2.2 Computation Offloading techniques are discussed and categorized based on various factors of relevance to the focus of this thesis. Using this taxonomy, the most relevant computation offloading technique (to the focus of this thesis) is identified, justified and discussed in more detail. The subsection concludes with a discussion of existing strategies for determining component-to-device placements and the limitations of using static approaches in resource dynamic mobile environments; thus setting the rationale for adaptive component placement approaches.

Section 2.3 follows with a discussion on Adaptive Computation Offloading as a means of addressing the limitations identified in section 2.2. It specifically focuses on adaptive component placement decision algorithms and examines the limitation of existing state-of-the-art approaches with regards to the software quality metrics of concern to this thesis (efficiency, scalability and decision efficacy) while deferring detailed technical analysis of existing approaches to the relevant future chapters, for brevity and better readability.

2.1 Application Adaptation

Wireless networks and mobile computing devices, which typically characterize mobile computing environments, introduce unique challenges to the design,
CHAPTER 2 Literature Review

implementation and execution of an application (Satyanarayanan M., 2001). The mobility, heterogeneity and resource constrained characteristics of computing devices coupled with the variability and unpredictability of the underlying communication networks, necessitate that applications adapt their behaviour so as to operate efficiently in dynamic and changing environments.

Application Adaptation refers to the ability of an application to alter its behaviour at runtime to better operate in a given execution scenario. For instance, an application might reduce its network communication in the event of a detected drop in bandwidth (Kim and Copeland, 2003), or it might spread out its memory consumption across multiple nodes due to lack of resources (Ryan C. and Westhorpe C., 2004).

Application Adaptation generally involves at least three essential processes: 1) Environmental awareness through collecting, aggregating and processing of information collected from sensors or devices. This information is used to infer or predict situations, which require an adaptation of the executing application. 2) Computing an Adaptation Decision, by determining the type or degree of application reconfiguration that must be performed in order to respond to the changes in an environment detected in step 1, and finally 3) Effecting the application reconfiguration decisions computed in step 2.

The first adaptation process, as identified above, will be discussed in section 2.1.1, followed by a discussion of Adaptation Decision Computation (step 2) in section 2.1.2. The last step of an adaptation process is implicitly discussed in the above two subsections as it does not directly relate to the research focus of this thesis, and does not present additional complexities that are not covered in discussions of the methodologies employed in the above two processes.

2.1.1 Context and Context-Awareness

The proliferation of sensors in mobile devices, and the availability of high-level APIs for querying and processing their data, is increasingly making environment aware application features commonplace in mobile applications (Location based services (Hofmann-Wellenhof et al., 1997; Kaplan and Hegarty, 2006) etc.). Such environmental awareness is fundamental to any adaptive system, which must monitor internal or external factors that influence an application’s behaviour, such as the user, the host device (Fox et al., 1996), the network infrastructure, and various other entities (Badrinath et al., 2000; Guanling and David, 2000).

In application adaptation, the raw data collected or inferred from environment sensors or adaptation middleware are generally termed as Context (Dey and Abowd,
2000a; Dey and Abowd, 1999) (Schmidt et al., 1999), and thus a system that monitors and utilizes such information is termed Context-Aware. Given the wide coverage of context under this definition the authors in (Bill N. Schilit et al., 1994) classified context information into three categories based on the type of context measured: 1) Physical context such as lighting, and temperature 2) User context, such as location or preference settings, and 3) Computing context, such as network bandwidth and device specific resource consumption information. In terms of context-awareness the authors in (Guanling and David, 2000) identified two categories, passive and active context-awareness, based on the response of a system to the change in context.

In passive context-awareness an application (or underlying middleware) delivers context information to the user (or persists it for later use), thus delegating any response to him/her whereas in active context-awareness a context change requires automatic reconfiguration or adaptation of an application’s behaviour. While the type of context utilized by Application Adaptation can fall into either of the three categories identified by (Bill N. Schilit et al., 1994), an active form of context-awareness is implicitly required by any adaptive application, and is assumed for the remainder of this thesis.

Measurement of context information can either be performed in a push based acquisition strategy whereby the application is directly notified of changes in context data, or a pull based strategy wherein the application is responsible for determining change in context by querying the underlying middleware or context provider (sensor devices etc.). The acquisition strategy employed by an adaptation system could differ for each monitored context and depends on both the type of measured context and the features provided by the context provider (sensor etc.). However, pull based approaches are more widely adopted as they do not rely on an underlying context provider to offer special push capabilities (Schwenger et al., 2005).

In the case of the pull based acquisition strategy, querying can be performed either infrequently in the case of static (or rarely changing) context, such as a device’s resource capacity (memory capacity, processing capacity etc.) or it could be performed continuously in the case of context that dynamically changes through time, such as the resource usage of a device or a user’s location. Often, dynamic context is more representative of the external factors for which an application must adapt. Hence such context must frequently be collected in order to obtain up-to-date information about the environment so as to ensure both accurate and responsive adaptation, while balancing the associated overheads of frequent context measurement (Gani H. et al., 2006).
CHAPTER 2 Literature Review

Continuous measurement of context creates a context-history which is used in some cases to analyse patterns and predict future behaviour (Mayrhofer, 2005). For instance, in the tourism guide applications discussed in (Cheverst et al., 2000; Hristova et al., 2003; O'Grady and O'Hare, 2004; Schwinger et al., 2005), the locations to which a user has travelled are recorded and analysed so as to recommend future destinations of interest to the tourist. Similarly (Narayanan et al., 2000) records history of the resource usage of applications on a mobile device so as to predict their future resource requirements on the specific device. This approach to context management not only enables proactive adaptation of an application to optimize for likely environmental changes (instead of reactively to current context) but can also be used to reduce the overheads of frequent context measurement discussed in the previous paragraph.

Once context information is collected, and a scenario in which an application must reconfigure itself detected, adaptation occurs. Such adaptations are classified based on the system’s lifetime in which they occur and hence could be either statically performed during compile or deployment time (offline), or dynamically at runtime (online) (McKinley et al., 2004).

Static (offline) adaptation approaches are typically performed once, and are employed in conditions in which the environmental settings are known or queried prior to execution of the application. While this approach can to some extent address heterogeneity of an environment and reduce or remove performance cost of adaptation during runtime, it assumes a static configuration and behaviour of the execution environment and hence does not cope with runtime dynamism of devices, applications, or external conditions such as user mobility. This makes the approach infeasible in mobile and pervasive spaces, on which this thesis is focused.

On the other hand, in online approaches, adaptation is continuously performed at runtime, based on dynamic context information. This allows an application to evolve and continue to optimize its behaviour under changing environmental conditions, making it suitable for resource dynamic environments such as mobile and pervasive spaces. Nevertheless, unlike offline approaches, online adaptation incurs runtime costs associated with the various processes of adaptation. Hence, despite detected context changes, such approaches only perform adaptation under cases in which either adaptation gains outweigh costs (decision computation cost discussed in section 2.1.2 below) or adaptation is critical (e.g. the application would otherwise need to terminate). For instance, an application that adapts by distributing its runtime components to external devices might only do so when an external device with
notably more computation resources is detected (this approach is discussed in section 2.3). Similarly, an application which adapts by optimizing its memory footprint, might choose to do so only when a device’s memory availability falls below a set threshold (e.g. 10%). Further discussion of context management from the context of Adaptive Computation Offloading is discussed in section 2.3.

2.1.2 Adaptation Decision Computation

Once context information about the environment has been collected, and the need for adaptation determined, an Adaptation Decision is computed. An adaptation decision computes the degree and specific type of application reconfiguration that must be performed under the new environment, in order to achieve a required functional or non-functional objective (discussed under section 2.1.3). Khan (Khan, 2010) categorizes adaptation decisions into the following decision reasoning strategies 1) Rule-based 2) Goal based, and 3) Utility based.

Rule-Based Adaptation: In rule based approaches (Efstratiou et al., 2002; Wu et al., 2008) adaptive responses are pre-specified for anticipated context changes. This mapping of context change to adaptation action is typically performed using policy languages. While such approaches allow for well-defined adaptation behaviour, they require mapping of low-level events rather than higher-level abstractions of contextual events which makes it harder to specify non-functional behaviour or QoS concerns (Khan, 2010). In addition, such approaches require specific mapping of all possible context changes, thus making the approach more difficult to scale and manage in resource dynamic environments (such as mobile and pervasive spaces) where complex and unforeseeable context changes and interactions are likely to be commonplace.

Goal Based Adaptation: In Goal based strategies (Cheng et al., 2009), adaptation is performed to achieve a specific desired state, with all other possible states being seen as undesirable (Khan, 2010). While the approach offers a higher-level decision making strategy than rule-based approaches, this binary view of adaptation is limiting for scenarios where optimal adaptation outcomes are either unknown or unattainable (Gu et al., 2003; Li et al., 2001; Ou et al., 2007b; Xiaohui Gu et al., 2004), and greater flexibility in adaptation outcomes is desired in order to negotiate or balance various objectives.

Utility Based Adaptation: in utility-based adaptation, acceptable solutions, rather than optimal ones, are sought by computing a small number of alternative configurations (primarily because of computation constraints) to an application; and
selecting the most feasible of the variants. This offers a greater degree of decision flexibility than either of the above approaches and is more applicable to scenarios where optimal adaptation outcomes are either costly to compute or intractable. Hence the strategy has largely been applied in adaptation approaches within mobile and pervasive environments (Noble and Satyanarayanan, 1999) (Khan, 2010) including those of interest in this thesis (Gu et al., 2003; Li et al., 2001; Ou et al., 2007b; Xiaohui Gu et al., 2004), as will be discussed in more detail in section 2.3 and Chapters 3-6. As such, for the remainder of this thesis, adaptation refers to utility-based adaptation unless otherwise explicitly stated.

While the above categories discuss general strategies for computing an adaptive decision, the greatest divergence and variation of existing work on Application Adaptation is on the type of behavioural reconfiguration performed on an application using such decision strategies. The following section aims to provide an overview of the major application adaptation variations so as to identify and rationalize the specific approach adopted in this thesis.

2.1.3 Adaptation Variations

Given the diversity of application behaviours and possible execution environments, existing research has explored the adaptation of different facets of software (code, design structure, protocol etc.) in order to optimize for a wide range of objectives (performance, resource utilization, security etc.).

In order to provide broad coverage of these strategies, this section categorizes existing work based on the following three factors of relevance to this thesis 1) Software Layer in which adaptation capabilities are enabled 2) Type of Application Behaviour adapted (Functional Vs. Non-Functional) and finally 3) Adaptation Mechanisms employed.

2.1.3.1 Adaptation Layer

As briefly discussed in the sections 2.1.1 and 2.1.2, implementing application adaptation involves enabling capabilities such as context management, and adaptation decision computation. Depending on the software layer in which these capabilities are implemented, the authors in (Jing et al., 1999; Satyanarayanan, 1996a, b) identify three types of adaptation strategies; 1) Laissez-Faire Adaptation, 2) Application Transparent Adaptation and 3) Application-Aware Adaptation.

In Laissez-Faire approaches, all adaptation capabilities such as context management and adaptation decision computation are performed by the application
itself, thus requiring these capabilities be developed as part of the application’s functional logic. For instance in (Ghadse, 2011) an audio recording application optimizes its battery usage based on the type of conversation being recorded, by optimizing the device’s behaviour accordingly.

While this approach allows for adaptive capabilities that are specifically targeted to the domain requirements of individual application, it incurs development and maintainability costs which make the approach problematic in scenarios where complex adaptation behaviour is required as would be the case for large applications in pervasive environments (Rossi P. and Ryan C., 2005). Furthermore, reuse of these capabilities for adapting other software is difficult due to the specificity of the adaptation to individual applications, or application domains. In addition, such approaches would not be applicable to existing non-adaptive applications, making this approach unsuitable for transitioning computationally heavy applications to mobile environments (which is the focus of this thesis as discussed in Chapter 1).

At the other end of the adaptation spectrum, is the Application Transparent approach, in which all adaptation capabilities and tasks are performed in an underlying middleware or operating system without the knowledge or involvement of the application. For instance in (Flinn et al., 2001), upon detecting a reduction in bandwidth, a middleware service systematically degrades the quality of content transmitted by a multimedia streaming application, without its awareness or involvement in the process. Unlike Laissez-Faire approaches, this reduces development complexity and cost, and allows the reuse of adaptation capabilities across different applications. However, the approach only provides generic adaptation capabilities and does not leverage domain specific behaviour which limits adaptation flexibility, thus making it unsuitable for the adaptation of mobile and pervasive applications within the context of this thesis (since such adaptation requires application specific information such as resource usage and coupling behaviour of components as will be discussed in greater detail in section 2.3.)

In-between these two extremes is Application Aware adaptation, in which an application cooperates with the underlying middleware to perform both generic and application specific adaptation. In such an approach, an application would typically provide its behavioural characteristics (runtime behaviour information, component design structure information etc.) to the underlying middleware, which uses it to compute an adaptation. While such approaches require the integration of additional capabilities into an application various works have proposed automatic (transparent)
injection of such capabilities with minimal or no development effort (Arun Mukhija and Glinz, 2005; Gani and Ryan, 2009; Ryan C. and Westhorpe C., 2004).

Hence, as will further be discussed in section 2.3, the approach employed in this thesis (Adaptive Computation Offloading) involves adaptation which must factor in application specific information during adaptation decision-making. Consequently, Application Aware Adaptation approaches are assumed for the remainder of this thesis.

2.1.3.2 Adaptation Mechanisms

Gani (Gani, 2010), classifies adaptation as either functional or non-functional, based on the type of application behaviour adapted. In functional adaptation, the functional behaviour or feature of an application is modified in response to environmental changes. For instance, a tourist guide application might offer different functionalities based on the location of the user i.e. when the user is near a cinema, the application presents him/her with movie session times and ticket booking facilities, whereas when near a historic site the user could be presented with factual information and interactive/augmented viewing of the surroundings (Mukhija and Glinz, 2005). Such forms of adaptation are typically implemented in component model (or service oriented) architectures, wherein the function [parameter adaptation (DeVaul and Pentland, 2000; Dey and Abowd, 2000b; Fickas et al., 1997; Flinn et al., 2001; Kortuem et al., 2001; Minar et al., 1999; Sousa and Garlan, 2002)] or structure [compositional adaptation (Aksit et al., 1992; Aksit and Choukair, 2003; Arun Mukhija and Glinz, 2005; Chen et al., 2001; Hiltunen and Schlichting, 1996; Mukhija and Glinz, 2005; Renesse et al., 1997; Venkatasubramanian, 2002; Vukovic and Robinson, 2004)] of components (or services) are modified in response to context changes.

In Parameter Adaptation (DeVaul and Pentland, 2000; Dey and Abowd, 2000b; Fickas et al., 1997; Flinn et al., 2001; Kortuem et al., 2001; Minar et al., 1999; Sousa and Garlan, 2002), variables within a component, which determine its behaviour, are modified so as to invoke changes in the operation of the component. An example of this approach is the Pupeteer (Flinn et al., 2001) middleware which adaptively distils multimedia content transmitted between a client and a server so as to optimize power conservation under various environmental conditions. For instance, degrading the resolution of image or video transmitted to a mobile device when detecting poor bandwidth. Parameter adaptation has also been implemented to enforce different Quality of Service levels, where multimedia delivery applications adaptively configure different levels of content quality based on various contextual factors, with
such approaches including image servers (Bolliger and Gross, 1998) and video streaming applications (Noble, 2000). While parameter adaptation offers fine-grained control over the behaviour of an application, it is limited in that it cannot introduce new behavioural strategies into an application in response to unanticipated environmental changes.

Another functional approach is Composition adaptation (Aksit et al., 1992; Aksit and Choukair, 2003; Arun Mukhija and Glinz, 2005; Chen et al., 2001; Hiltunen and Schlichting, 1996; Mukhija and Glinz, 2005; Renesse et al., 1997; Venkatasubramanian, 2002; Vukovic and Robinson, 2004) in which the component structure of an application is altered by adding, removing or replacing one or more components with alternative versions that offer different functionality. Unlike parameter adaptation approaches, this allows the introduction of new capabilities in response to environmental changes, as in the case of the tourism guide application discussed earlier. For instance in Fractal (Lufei and Shi, 2005) the authors propose a framework, which dynamically optimizes communication by selecting from a number of alternative application level protocols (GZip, direct sending etc.) based on changing device and network conditions. While these approaches have primarily been used for adapting functional concerns, they have also been utilized to optimize non-functional behaviour. For instance, McKinley and Padmanabhan (McKinley and Padmanabhan, 2001) proposed an approach for dynamically inserting an error corrective component (as pluggable java I/O Stream classes) to improve reliability of an audio and video streaming application in the event of degrading wireless connectivity. However, functional adaptation strategies in general incur two major limitations. Firstly, such approaches often require implementation of alternative behaviours, which increases development and maintainability cost. Secondly, adaptations are tightly coupled to the domain requirements of individual applications and are thus not generically applicable to other domains, which further increases cost and reduces adaptation flexibility and reusability.

Non-functional (extra-functional) adaptation approaches on the other hand, try to optimize various non-functional concerns such as security, reliability, performance, fault-tolerance etc. without affecting or altering the functional requirements or behaviour of an application. Unlike functional approaches, non-functional strategies are not tied to specific application implementations and can generically be applied to a wide range of applications, while factoring in application specific behaviour (see Application Aware adaptation in section 2.1.3.1). The most prominent examples of such approaches include Adaptive Component Replication and Adaptive Component
Mobility strategies. In Component replication (Katmon and Ryan, 2011; Marin et al., 2007; Strauss and Theel, 2003), copies of runtime components of an application (as objects, classes, services etc.) are created and managed across multiple devices so as to improve reliability and fault tolerance and in some cases application performance (Katmon and Ryan, 2011). While the approach is beneficial for mobile and pervasive spaces where device disconnection or failure is commonplace, it requires additional overheads associated with running, updating (synchronizing) and managing component replicas and is hence not considered ideal for adapting computationally heavy applications in mobile spaces.

On the other hand, in Adaptive Computation Offloading (Adaptive Computation Mobility) (Abebe and Ryan, 2009; Gani H. et al., 2006; Ou et al., 2006; Rossi P. and Ryan C., 2005; Ryan C. and Westhorpe C., 2004; Violeta Felea and Toursel, 2004; Xiaohui Gu et al., 2004) an application distributes its runtime components to one or more devices so as to execute computationally heavy applications by leveraging externally available resources. Unlike component replication strategies, the approach does not introduce additional application components but instead optimizes existing component localities thus incurring less adaptation overhead. In addition, the approach allows for increased flexibility of adaptation outcomes through various potential component-to-device distributions. Hence, this strategy is adopted in this thesis, and is discussed in the remainder of this literature review. Specifically, section 2.2 first introduces the various computational mobility strategies and identifies and rationalizes the specific type focused on in this thesis. Sections 2.3, discusses adaptive approaches to computation mobility and their utility in resource dynamic environments. The subsection specifically focuses on existing works and their limitations with regards to software quality attributes such as efficiency, scalability and decision efficacy, which this thesis aims to address.

2.2 Computation Mobility

Fuggetta et al. (Fuggetta et al., 1998) define Code Mobility (Computation Mobility) as the transfer of computation units from one execution environment (e.g. device) to another. In this context, computation unit refers to code abstractions ranging from coarse grained processes (D. Milojicic et al., 2000), which abstract the execution of an application in an operating system, to fine-grained atomic variables and code statements within an application module (Jul et al., 1988; Mascolo et al., 1999). Code mobility is performed for a number of objectives such as load balancing (D. Milojicic et al., 2000; Douglis, 1990; Rossi P. and Ryan C., 2005), performance
improvement (Hütter and Moschny, 2008), reliability (Katmon and Ryan, 2011), and power conservation (Li et al., 2001).

Computation mobility can be of either stateless or stateful form (Gani, 2010). In Stateless code mobility, only code modules are transferred to a remote machine for execution, after which point a client device (recipient of code) creates a new instance of the application or component. Common examples of this approach include Java Applets (Sun Microsystems Inc., 1995), VBScript (Microsoft Corp., 1996) and JavaScript (Netscape Corp., 1995) modules, which are downloaded from a remote webserver onto a client’s browser for execution. On the other hand, in stateful code mobility, the state of an execution unit (e.g. object) such as its variable data, program counters etc. are transferred along with code modules (Cabri et al., 2000). This allows a running application to move and distribute its components without affecting the flow of execution. Typically in such an approach the transfer of a computational unit is performed in order to optimize the runtime behaviour of an application (both functional and non-functional) rather than to deliver functional capabilities to a device. Hence, only stateful code mobility approaches are applicable to the focus of this thesis, and are discussed in the remainder of this section.

Depending on the degree and type of state transfer allowed, stateful code mobility can be classified as either strong or weak mobility (Fuggetta et al., 1998). In strong mobility the migration of a computation unit entails the deep transfer of its execution state such as its execution stack and register information. Since the approach requires the transfer of device specific state information it is difficult to implement across heterogeneous processor and memory architectures (Barak et al., 1993; Douglis, 1990). Consequently, research efforts (Barak et al., 1993; Jul, 1989) have focused on strong mobility in homogenous collaborations, under specific execution platforms while fewer works have looked at enabling migration of threads across different architectures by using a virtual machine abstraction layers (Jin et al., 2009).

On the other hand, in weak mobility only the data state (attribute values etc.) of a computation unit is migrated along with its code modules, and hence after migration a client device is responsible for reconstructing a component’s state so as to resume execution. Typically, such approaches offer portability across heterogeneous environments as they do not rely on lower-level hardware details and delegate the responsibility of reconstructing an application’s state to an underlying middleware. Most computation mobility approaches including those focused on in this thesis (as will be discussed in the following subsection) employ weak mobility techniques and hence this technique is assumed for the remainder of this thesis.
2.2.1 Computational Units of Mobility

Computation mobility has been performed with execution units ranging in granularity from Virtual Machines (Jin et al., 2009) to individual atomic statements inside code modules (Jul et al., 1988; Mascolo et al., 1999). The mobility granularity largely indicates the objectives of migration and the execution domain in which it is applied, and hence this subsection aims to discuss some of the common computation mobility granularities and identify the specific type of mobility focused on in this thesis (object mobility), which is discussed in detail in section 2.2.1.1.

Virtual Machine Migration: refers to the migration of software emulating a live (running) virtual device from one physical host machine (source) to another (destination) (Jin et al., 2009; Nelson, 2010). This migration involves the transparent mobility of a running operating system and all associated services and applications; which are shielded from the complexity of the migration and the heterogeneity of the host platform. The approach is often used for improving performance, availability and load balancing of servers or applications in cloud infrastructure such as the Amazon Elastic Compute Cloud (Amazon Inc., 2006; Voorsluys et al., 2009).

Service Migration: In service-oriented architecture, a service is a set of related software functionalities involving multiple modules or classes. Hence Service migration involves the mobility of these functionalities in terms of either their classes (Ou et al., 2007a; Ou et al., 2006; Ou et al., 2007b) or OSGi modules (Rellermeyer et al., 2007) to remote devices. The approach is used for optimizing various objectives, in Grid, Cloud and recently Mobile computing settings (Rellermeyer et al., 2009) (Zhang et al., 2010b). Such migration often requires the intermediation of middleware to abstract the heterogeneity of underlying devices. Despite the primarily stateless nature of services, most work (Li et al., 2001; Ou et al., 2006; Rellermeyer et al., 2007) has explored stateful migration capabilities in order to enable the migration of live services.

Process migration: Involves the mobility of processes, which are an operating system’s abstraction of an executing application and its controlled resources. Most process migration involves strong mobility over homogenous settings, and is often performed at either an operating system level (Barak and Shiloh, 1985) (Walker and Mathews, 1989) (Douglis and Ousterhout, 1991), Microkernel level (Theimer et al., 1985) or User level (Litzkow and Solomon, 1992) and involves the migration of all associated threads owned by the process. While classically, the approach was implemented for mobile agent systems (Adnan et al., 2000), which are autonomous, mission driven entities, additional works have focused on process migration for

**Thread Migration:** A thread represents the simplest unit of execution, which is schedulable by an operating system. Unlike the coarser approaches discussed above, thread migration allows for finer control over parallelism, locality and load balancing of parallel applications in cluster and grid computing environments. Most works in thread migration rely on Java and its associated technologies (RMI, Serialization etc.) for portability in heterogeneous environments. Hence these approaches employ weak mobility techniques (Felea et al., 2004) (Fahringer, 2000) (Michael Philippsen and Haumacher, 1998) (Philippsen and Zenger, 1997) due to the inherent lack of support for strong mobility in the JVM, while few works have explored explicitly capturing a thread’s state to enforce strong mobility (Truyen et al., 2000).

Mobility using the above approaches offers coarse-grained control over the distribution of computation, and in the case of service migration, assumes specific software architecture. This makes such approaches ideal for adapting applications in cluster or grid computing environments but less pertinent for mobile spaces. This is because the limited resources of mobile devices means that the amount of resources offered by individual devices could be low and thus the components placed on them have to be of finer and lighter footprint. Hence, fine-grained stateful mobility of execution units is requisite in order to enable the migration of computationally heavy applications in mobile spaces (which is the focus of this thesis). Object Migration is an approach that offers such fine-grained control over migration and is thus the focus of this thesis.

### 2.2.1.1 Object Migration

In the object-oriented paradigm, a runtime instance of a class (object) is the atomic unit of an application’s execution. In object migration (object mobility), an application’s objects (and if necessary their class files) are migrated to different machines for a number of objectives. In early works, object mobility was utilized for the mobility of software agents, which are autonomous mission driven entities that migrate to access goal requisite resources within a network. Recently however, works on object mobility have focused on distributing traditional applications across devices (Abebe and Ryan, 2011a, b; Fahringer, 2000; Felea et al., 2004; Gani H. et al., 2006; Ou et al., 2006; Ou et al., 2007b; Philippsen and Zenger, 1997; Rossi P. and Ryan C., 2005) for various objectives including load mitigation (Abebe and Ryan, 2011a, b), load balancing (Rossi P. and Ryan C., 2005; Ryan C. and Westhorpe C., 2004),
Facilitating object mobility within an application requires additional behaviour such as remote invocation, object location tracking, and the collection of various object related context information (discussed in section 2.3). Such capabilities often require the support of underlying middleware with which an application must interact to enable mobility. Various works have focused on reducing the associated development effort or cost through the automatic injection of such capabilities, with varying degrees of transparency. For instance Fargo (Gazit et al., 2000) allows automatic compilation of mobility and object monitoring (context information collection) capabilities but requires developers to restructure or redesign applications to an architectural model of object-groups known as Complets. On the other hand, MobJeX (Gani, 2010; Gani and Ryan, 2009; Gani H. et al., 2006; Rossi P. and Ryan C., 2005) provides automatic injection of mobility capabilities through manually specified annotations on classes. Compared to Fargo this approach minimizes the associated development effort as it does not require a re-design or re-development of existing applications in order to enable mobility capabilities, but still requires a degree of developer involvement. As discussed in Chapter 1, the aim of this thesis is the seamless transition of computationally heavy applications to mobile spaces, which necessitates low development cost/effort and hence the use of the most transparent approach (the approach by MobJeX (Gani, 2010; Gani and Ryan, 2009; Gani H. et al., 2006; Rossi P. and Ryan C., 2005)) is assumed for the remainder of this thesis.

Object mobility systems have generally been applied to two separate application paradigms; parallel and sequential applications. The objectives of these two domains are different and hence present different challenges. Object mobility for parallel applications [9, 14, 25] has focused on cluster and grid computing environments with objectives such as data and thread locality. JavaSymphony (Fahringer, 2000), JavaParty (Philippsen and Zenger, 1997), and ADAJ (Felea et al., 2004) are examples of such middleware targeting parallel applications in which objects are mobilized based on their involvement and usage by different threads or activities (tasks) with the aim of either collocating related threads to minimize remote calls, or distributing unrelated ones to reduce resource contention, increase parallelism and consequently improve application performance.

On the other hand, works on sequential application adaptation have focused on more diverse computing environments including pervasive spaces with objectives...
such as performance improvement, load mitigation and power optimization, and are hence more relevant to the focus of this thesis. Object mobility in sequential applications presents more challenges due to less explicit division in the units of distribution, and thus must instead focus on minimizing inter-object network communication while maximizing utilization of external resources.

In object mobility for sequential application, two different types of objects are identified, stationary objects and mobile objects. Stationary objects are tied to a specific device and cannot be migrated without breaking the functional requirements of an application. This could be because they either access device specific resources such as files, local databases and sensors (GPS, camera, etc.) or provide user interface components. On the other hand, mobile objects can freely be migrated between networked devices without affecting the functional behaviour of an application. The placement of such objects directly influences the non-functional properties (performance, resource utilization, power consumption etc.) of an application and has thus been the subject of various research.

Existing work has focused on either manual (user controlled) or static analysis based placement of mobile objects to devices. For instance, (Philippsen and Zenger, 1997) employed a hardcoded scheme of object placement in which a programmer specifies (at development time) where each object instance is to be placed within a network. Similarly (Tilevich and Smaragdakis, 2002, 2009) employ an offline static application partitioning scheme to determine which object groups to place to which devices prior to executing an application. In comparable work Fahringer (Fahringer, 2000) proposes an approach in which a networked environment is statically modelled as a virtual node tree to which a developer must specifically map objects during development time.

While such approaches might be applicable to fixed homogenous collaboration environments executing stable and predictable applications, they are not applicable for mobile or pervasive spaces. This is because the dynamic behaviour of such environments (device disconnection, resource availability fluctuations, user mobility, application resource usage variability etc.) would render static placements either redundant or suboptimal as the environment changes.

Consequently, other work has focused on adaptively reconfiguring the distribution (or placement) of objects at runtime based on environmental changes. This allows an application (and the underlying middleware) to continue optimizing non-functional concerns (performance, resource utilization, power etc.) under
CHAPTER 2 Literature Review

changing environmental conditions; which is a requirement of this thesis and thus discussed further in the following subsection.

2.3 Adaptive Computation Offloading

As discussed in the previous section, variability in mobile environments and application behaviour can result in the sub optimality or infeasibility of an existing object distribution within a collaboration environment. For instance, a device might run out of resources (e.g. power, memory etc.) or a client might move to an area with reduced network connectivity, thus being unable to host the same mobile objects as it did prior to the event. Such dynamic environments require that a strategy dynamically reconfigure object distributions so as to maintain execution of an application and preserve its fidelity while minimizing the cost to the collaboration.

Adaptive Computation Offloading is such a strategy, in which components are dynamically allocated and reallocated to devices based on environment changes, in order to optimize a number of objectives. While the term generally applies to any type of execution unit (process, thread, objects etc. as discussed in section 2.2.1), this study is specifically concerned with object mobility based adaptation and hence the term Adaptive Computation Mobility in this thesis refers specifically to Adaptive Object Mobility strategies.

As is the case with any adaptation strategy (as discussed in section 2.1), Adaptive Computation Offloading involves two components: 1) A context collection and management process, responsible for monitoring the environment (devices, network connectivity etc.) and application behaviour (object resource utilizations and coupling patterns etc.) as will be discussed in section 2.3.1 below and 2) An adaptive decision computation component, which is responsible for determining new object-to-node placements, which occurs when sub-optimality of old distributions is detected by component 1 above. This computation component is discussed further in sections 2.3.2 and 2.3.3.

2.3.1 Context Management for Adaptive Computation Offloading

As discussed in section 2.1, computing an adaptation decision requires an awareness of the environment. In Adaptive Computation Offloading, this context includes the resource requirement and interaction behaviour of application objects, as well as the resource availability of devices and the underlying network infrastructure.
This information allows an adaptation system to both detect environmental changes which merit adaptation, as well as determine the resource constraints under which an adaptation decision must be computed (as will be further discussed in sections 2.3.2 and 2.3.3).

In Adaptive Computation offloading, such context measurements are termed Metrics and can be of either base or derived types (Gani, 2010). The former refers to contexts that are directly measurable (from a sensor, device or underlying middleware) such as the memory capacity and memory usage of a device, whereas derived metrics are calculated from one or more metric values (either base or derived). An example of a derived metric could be the response-time of invoking a remote method, which in (Gani H. et al., 2006) is calculated as the sum of the in-processor time of the method on the remote device and the method invocation network latency (due to transmitting parameters and receiving return values).

There are various strategies for collecting, aggregating and disseminating context (metrics) in Adaptive Computation Offloading, which have been outlined in the context of general adaptation strategies in section 2.1.1 and are hence not discussed again in this section. In general, metrics management in Adaptive Computation Offloading ensures that an adaptation decision utilizes the most up-to-date information about the environment (through continuous runtime metrics collection) while leveraging metrics history to compute decisions that are based not only on temporal environmental settings but indicative of potential or future changes (section 2.1.1. context history and context aggregation).

While in section 2.1.1 a taxonomy of context data for general adaptation strategies was identified, for the case of adaptive object mobility, Rossi and Ryan (Rossi and Tari, 2007) (Ryan C. and Rossi P., 2005) proposed two context type categories: a) Environment metrics b) Software metrics. Environmental metrics measure context pertaining to resource availability and resource usage of devices and their underlying network infrastructure. These include measurements of the network, memory, processor and power interfaces of devices as identified in (Rossi and Tari, 2007) (Ryan C. and Rossi P., 2005) and summarized in Table 2-1. While the resource capacity of a device (memory capacity, network capacity etc.) is static and need only be measured once (or specified by a developer), the resource utilization metric is dynamic and must be frequently measured at runtime. Similarly, resource availability metrics are derived from these two metrics and must hence be computed frequently to assess resource constraint within a device.
Table 2-1 Environmental resource usage metrics proposed by (Rossi and Tari, 2007)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Capacity</td>
<td>The memory capacity of a device</td>
<td>bytes</td>
</tr>
<tr>
<td>Processor Capacity</td>
<td>The processor capacity of a device</td>
<td>instructions/s</td>
</tr>
<tr>
<td>Network Capacity</td>
<td>The bandwidth limit of a device</td>
<td>bytes/s</td>
</tr>
<tr>
<td>Memory Utilization</td>
<td>The total memory usage of a device</td>
<td>bytes</td>
</tr>
<tr>
<td>Network Usage</td>
<td>The network utilisation of a device</td>
<td>bytes/s</td>
</tr>
</tbody>
</table>

While several works have defined software metrics to quantify the design quality and runtime behaviour of traditional software, they are not directly applicable to the specific requirements of adaptive object migration strategies. In adaptive object migration, software metrics measure the runtime resource usage (processor, memory, and network) and interaction behaviour (coupling) of an application’s objects so as to determine the resource requirements of an application as well as to compute the optimality of potential object distributions.

Of existing literature the most explicit discussion of software metrics measurement is that provided by Rossi and Ryan (Ryan C. and Rossi P., 2005), in which the resource usage of components is determined using smaller base metrics, such as the execution time of methods, and the network invocation cost of a remote method. To a large extent, their metrics employ device agnostic units such as bytes for network and memory usage, and number of executed instructions for processor utilization, so as to enable better prediction of the resource utilization of alternative object distributions.

However, while the work by Rossi and Ryan (Ryan C. and Rossi P., 2005) considered various metrics for determining the usage history of an object’s methods (Size of Serialized Parameters, Number of method Invocations etc.), they did not explicitly consider coupling metrics for the interaction between objects which is of particular importance in determining the optimality of object distributions. The greater the inter-dependence of two objects the higher the remote invocation cost of placing them apart (network, power, performance), thus influencing the decisions performed by adaptation algorithms as will be discussed in section 2.3.3. To address this limitation (Gu et al., 2003; Xiaohui Gu et al., 2004) defined metrics which quantify inter-object coupling as the number of method invocation between any two objects, where higher method invocation counts signified relatively high coupling between two objects. However, this approach does not accurately represent the true resource cost of placing components apart. For instance while the inter-object
invocation count between two objects might be small, the amount of data passed during invocation as quantified by the size of the parameters and return type of invoked methods might be relatively larger than object pairs with higher invocation counts. As the factor of importance in adaptive decision computation is the amount of network cost that would result if any two objects were placed apart, recent work (Gani, 2010) proposed an approach for measuring coupling between objects as the total size of serialized parameters passed during method invocations. Consequently, the metrics suite by Rossi and Ryan (Ryan C. and Rossi P., 2005), and the object coupling metrics proposed in (Gani, 2010) are adopted in this thesis.

Once context information is collected, a context management component must determine whether an adaptation decision is merited. This decision must ensure the agility, or timely trigger, of adaptation in order to limit object topology inefficacies, while reducing the potential overheads of frequent adaptations. In (Abebe and Ryan, 2011a, b; Gani, 2010), a simple threshold scheme is used, in which an adaptation is triggered when the measured context exceeds a specified value; for instance, when memory utilization exceeds 80%. On the other hand (Gu et al., 2003) propose a fuzzy logic control model which offers increased flexibility on triggering conditions. Their work showed reduced bandwidth consumption and improved performance over fixed threshold approaches.

### 2.3.2 Adaptation Decision Computation

Once a decision to compute an adaptation is reached, a device computes a new object topology (object to device distribution) that optimizes for a number of objectives. Unlike parallel applications where threads determine a unit of separation, sequential applications do not have a clear functional separation of objects. Hence, the process of computing an optimal object topology involves determining the co-locality of individual or groups of objects based on resource availability in the environment, resource usage of the objects, and the coupling patterns between objects. This allows for the optimization of various objectives or concerns such as load mitigation, load balancing, performance or power usage. While in this thesis, generic adaptation objectives which can later be extended to encompass more objectives are considered, the evaluation scenarios considered in Chapters 3-6 typically focus on load mitigation for simplicity and clarity.

Based on the location in which adaptation decisions are computed, two different types of adaptation are identified in (Ryan C. and Rossi P., 2005): a) Global or centralized adaptation and b) Local or decentralized adaptation.
In Global or Centralized Adaptation, decision-making is performed by a single dedicated and resource copious machine. Other collaborating devices periodically communicate their environmental and software metrics to this device in order to provide it with the requisite context information for computing an adaptation. Once a decision is computed, the device implements the new object topology by triggering object migrations on collaborating devices. Given that the centralized device would maintain a complete model of the collaborating device and the executing application (through context information), it can theoretically compute a near optimal placement of objects-to-nodes. However the approach presents various limitations: Firstly, the approach is costly as a result of the communication of software and environmental metrics, and the computationally intensive process of determining an optimal component topology (Ryan C. and Rossi P., 2005). Secondly, the approach requires a dedicated resource copious server, which is not always possible in ad-hoc collaborations in mobile environments and, in addition, presents a central point of failure.

In Local or Decentralized Adaptation, decision-making is computed on individual collaborating devices. Typically, the resource metrics of a device are communicated throughout a collaboration environment whereas the propagation of software metrics differs based on existing works (Gani H. et al., 2006; Gu et al., 2003; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004). For instance (Gu et al., 2003; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) require information collected about components to be disseminated to external devices, whereas in (Gani H. et al., 2006; Rossi P. and Ryan C., 2005; Ryan C. and Rossi P., 2005) this information is of pertinence only to the local device. When a node runs out of resources, it employs a heuristic approach for computing an adaptation decision based on the information it maintains about the collaboration and the metrics of the objects in its memory space as well as the partial or complete view it might maintain about remote objects. The primary advantage of local adaptation is that it removes the central point of failure and allows for greater scalability while also reducing the cost of computing an object topology (Rossi P. and Ryan C., 2005).

Because of the limitations of a central point of failure in global approaches, this thesis focuses on local adaptation decision computation approaches. In these approaches a device computing an adaptation decision follows two steps: A) Selecting a set of candidate devices which can offer an optimal set of resources, and B) Computing an object to device mapping (object topology) that satisfies a number of desired objectives. A discussion of each sub-process is provided below so as to
provide the requisite background, and identify the existing limitations, which are to be addressed in this thesis.

2.3.2.1 Candidate Device Selection

The first step in computing an Adaptive Computation Offloading decision is for an adapting device to select a set of candidate devices to adapt to. Existing works fall under two different categories in this regard; the first category (Gu et al., 2003; Li et al., 2001; Ou et al., 2007a; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) involves works in which the adaptation target is a simple resource copious server. In this case, the server is assumed to have unlimited resources and hence the simple act of discovering it is sufficient for adaptation. Such works primarily focus on offloading to cloud computing resources and utilize basic service discovery techniques such as UPnP (Jeronimo and Weast, 2003) and Jini (Arnold et al., 1999).

However, the simplified client-server approach that is adopted in these works relies on the existence of a dedicated resource copious device (and a persistent connection to it), which might not be assured in dynamic and ad-hoc mobile computation environments. Hence, these works are not directly applicable to peer-to-peer based approaches (which are the focus of this thesis), in which the utility of each device within the collaboration needs to be evaluated for optimality before selection. While the underlying object topology computation strategies of these works can be considered, as will be discussed in Chapters 3-5, the device discovery approaches are not relevant to this thesis.

In contrast, works such as (Gani and Ryan, 2009; Gani H. et al., 2006; Rossi P. and Ryan C., 2005; Ryan C. and Westhorpe C., 2004) have focused on peer-to-peer adaptation scenarios in which the candidate devices are selected based on a number of criteria including their resource availability, bandwidth connectivity to the adapting device, and resource requirements of the adapting application. The most detailed of these works is (Rossi P. and Ryan C., 2005), which discusses a candidate device selection strategy which is tightly coupled with its object-topology computation process. In this approach, the utility of a remote device is computed based on the potential performance gain and load balance that would be attained by placing each runtime object to the device. Unlike the client-server model in which the server is assumed to have boundless resources, in peer-to-peer environments any device could run out of resources and require an adaptation. Hence, each device must frequently communicate its metrics to the rest of the collaboration so as to ensure that devices have the requisite information for the device selection process. However, in the context of pervasive spaces this approach has various limitations:
1) With an $O(EN^2)$ network communication complexity, where $E$ is the size of the stored environment metrics, and $N$ is the number of devices, frequently communicating environmental metrics (which is requisite to ensure accurate adaptation in resource dynamic environments) is costly (in terms of network, power and performance) and unscaleable with regard to collaboration size and the amount of context information considered (which is primarily influenced by the heterogeneity of the collaboration).

2) Storing collaboration-wide environmental metrics on each device presents memory overheads on constrained devices. With an $O(EN^2)$ space complexity where $E$ is the size of the stored environment metrics, and $N$ is the number of devices, the approach is unscaleable with regard to the number of collaborating devices or the amount of context information considered (again largely influenced by the heterogeneity of the collaboration).

3) The process of determining the utility of each device, based on the objects hosted on an adapting client, is computationally expensive. With a running time complexity of $O(NM^2)$, where $N$ is the number of devices and $M$ is the number of mobile objects, decision making is computationally expensive for mobile devices in this environment, exacerbated by the fact that these decisions are computed on already constrained (adapting) devices. In response, nodes need to set lower constraint thresholds for triggering adaptation so that enough resources to compute adaptation decisions are reserved. This lowered threshold in turn results in increased adaptation throughout the collaboration causing additional network overheads and reduced application performance.

In summary, while the approach in (Rossi P. and Ryan C., 2005) is feasible in small, stable, homogenous collaborations, the above constraints (1-3 above) limit the efficiency, scalability and overall feasibility of the approach under medium to large scale, heterogeneous and dynamic pervasive collaborations.

Hence, in chapter 3, this thesis addresses these limitations by reducing the resource and performance costs associated with the device selection process. To this end a novel distributed approach to candidate device selection is proposed, in which, upon request by an adapting node, each device is responsible for evaluating its own utility. An evaluation shows the approach reduces resource costs and improves performance in comparison to the existing approach (Rossi P. and Ryan C., 2005).
2.3.2.2 Computing an Object topology

Once a suitable subset of devices is selected, an adapting device determines a new object topology, which must (at a minimum) satisfy the following four objectives:

**Objective 1: Meet the resource requirements of all objects**

For a distributed and adaptive application to execute with minimal loss of fidelity (Performance, User experience etc.), its individual runtime objects must be placed on devices that are able to fulfil their requisite resources (memory, network bandwidth etc.) and maximize their efficiency.

**Objective 2: Achieve the requirements of the adapting client device**

Since in local adaptation, decisions are computed reactively in response to specific device related events such as a device running out of resources, the decision must address the cause of the adaptation by, for instance, reducing the load mitigation on the client device (e.g. reduce memory load by 20%).

**Objective 3: Limit migrations to the resource provisions offered by the remote candidate.**

Collaborating devices, which volunteer their resources to an adapting node, must often provide constraints that either align with their maximum resource availability or abide by user specific constraints (e.g. offer only 20% of free memory to an adapting device). An adaptation decision must ensure that a collaborating device is not provisioned with objects that exceed the resources it offers.

**Objective 4: To minimize the total inter-node network costs that would result from placing objects apart.**

An inevitable overhead resulting from the object distribution of sequential applications is the inter-device network communication that results from remote method invocations amongst objects. Minimizing this cost directly translates into improved application performance (by reducing network latency of remote method invocations) and efficiency of the adaptation outcome (quality of adaptation). To achieve this, a decision algorithm must ensure that highly coupled objects, which in this context refer to objects which have large method invocation overheads, are co-located on the same device.
2.3.3 Adaptive Object Topology Computation

An adaptive object topology computation decision, factors in both environment and software metrics to compute an object topology that optimizes the above four objectives. Though the unit of migration are objects, this decision is computed on abstract representations of their runtime behaviour, which can either be of object-level (fine) granularity (Gani and Ryan, 2009; Gani H. et al., 2006; Katmon and Ryan, 2011; Rossi P. and Ryan C., 2005; Ryan C. and Rossi P., 2005; Ryan C. and Westhorpe C., 2004) or class-level (coarse) granularity (Gu et al., 2003; Li et al., 2001; Ou et al., 2007a; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004).

2.3.3.1 Object-level Granularity

In object-level granularity adaptation (Gani and Ryan, 2009; Gani H. et al., 2006; Katmon and Ryan, 2011; Rossi P. and Ryan C., 2005; Ryan C. and Rossi P., 2005; Ryan C. and Westhorpe C., 2004), the locality of each individual object is decided independently by computing the utility of its placement on each of the available collaborating nodes during an adaptation. The most detailed example of object-level adaptation approach is that of Rossi and Ryan (Rossi P. and Ryan C., 2005) in which a score for each object-to-node matching is computed based on objectives 1-3 discussed in section 2.3.2.2.

Figure 2-1 shows pseudo-code for the basic decision making computation of a local adaptation algorithm computed by an adapting device as presented by Rossi and Ryan (Rossi P. and Ryan C., 2005). The evaluate() function computes a score determining the suitability of placing each mobile object, o on each remote target node, n. The object-to-node match with the highest score is selected for migration. The process is repeated until either all objects are migrated or an object-to-node match that can achieve a minimum threshold is no longer available. The score of an object placement (placement of object o to node n) considers two objectives: resource offloading from the constrained node, and performance improvement of the application. The resource offloading score evaluates the degree to which the load of the source node can be mitigated and the load difference with the target node reduced whereas the performance score evaluates the degree of response time improvement achievable through the migration of an object to a target. The approach can be readily extended to include more diverse goals such as battery life preservation, reliability etc. The score computation process as presented by Rossi and Ryan (Rossi P. and Ryan C., 2005) is discussed in more detail in Chapters 3 and 4.
do
  \{ 
  maxScore = 0.5 
  maxObject = null, maxNode = null 
  for each mobile object \( o \) in local node do 
    for each remote node \( n \) do 
      score = evaluate(\( o \), \( n \)) 
      if (score > maxScore) then 
        maxScore = score 
        maxObject = o 
        maxNode = n 
      end if 
    end for 
  end for 
  if (maxScore > 0.5) then 
    move maxObject to maxNode 
  end if  
\} while (maxScore > 0.5)

Figure 2-1: Local Adaptation Algorithm, Rossi and Ryan (Rossi P. and Ryan C., 2005).

Object-level adaptation approaches offer flexibility of object distributions which in turn allows for a more accurate matching of object resource requirements to external resource availability, hence giving rise to topologies that better satisfy the requirements of the application objects (objective 1), the client device (objective 2), and the collaborating external candidate devices (objectives 3). In addition, this allows for highly configurable mobility flexibility for individual objects of an application.

However, for reasons of computational feasibility (discussed further in Chapter 4) existing object-level granularity approaches omit coupling information from the decision making process thereby potentially reducing the co-locality of highly coupled objects and discounting adaptation objective 4 discussed earlier. The resultant object topologies thus incur higher inter-node communication limiting the applicability of this approach in pervasive or mobile spaces where network bandwidth is expensive and often constrained. Moreover, efforts to improve the quality of object-level adaptation by factoring in object coupling information result in computationally infeasible solutions as will be discussed and demonstrated in this thesis in Chapter 4. In addition, the process of computing the utility of each object’s placement on every device is computationally expensive, and the process unscaleable for heavy applications with larger number of application objects.

In summary, while object-level decision computation provides flexibility in determining object distributions, it incurs scalability limitations, efficiency costs, and results in suboptimal decisions as a result of the omission of object-coupling information.
2.3.3.2 Class-Level Granularity

To address the limitations of object level granularity adaptation, various works (Gu et al., 2003; Li et al., 2001; Ou et al., 2007a; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) have proposed the use of a coarser, class-level approach to decision computation. In such approaches, application objects are grouped and abstracted based on their class type. Hence, the runtime resource utilization and coupling behaviour of all objects of the same class type are represented as an aggregate. Specifically, the runtime of an application is represented as a dynamic weighted undirected class graph, G(V,E) where each vertex in the graph represents a class and all its runtime objects, whereas an edge represents the coupling between all objects of two class types.

Both vertices and edges are weighted, with different solutions using different schemes to represent the weight of vertices. For example, (Gu et al., 2003; Xiaohui Gu et al., 2004) represent vertex weight as the memory utilization of a class whereas Shumao et al. (Ou et al., 2006; Ou et al., 2007b) propose the use of a composite weight scheme which factors in the network, memory and processor utilization of a class into a unified vertex weight. On the other hand, edge weight is commonly represented as the total invocation count between all methods of the two classes, which only crudely approximates the true cost of coupling between classes, and thus a more accurate approach is proposed and used in chapters 4 and 5. The class graph (vertex and edge set, and their weights) evolves and dynamically updates to reflect changes to the application’s execution behaviour based on collected software metrics.

Given this dynamic application graph representation, object topologies can be computed by employing a heuristic graph-partitioning algorithm. Graph partitioning is the process of separating a graph G, into k number of disjoint subsets, each satisfying given constraints, while minimizing the number and total weight of edges whose ends fall between partitions; also known as the edge-cut of the graph. The problem is known to be NP-Complete (Garey and Johnson, 1979), but due to its diverse applicability in domains such as VLSI (Mead and Conway, 1980) and circuit design, various classic heuristics have been proposed (Chinthapanti, 2004; Fjallstrom, 1998; Hogstedt et al., 2001; Karypis and Kumar, 1995; Karypis and Kumar, 1996a, b). Nevertheless such heuristics are still comparatively resource intensive (Ou et al., 2006) and thus more suited to static graphs in resource copious environments.

Consequently, various works in adaptive object migration have adopted and modified the classic approaches for dynamic graphs in resource-constrained environments. For example, Gu et al (Gu et al., 2003) proposed an algorithm derived
from the min-cut (Stoer and Wagner, 1997) approach, for adapting between a constrained device and a dedicated surrogate (resource copious machine). The approach involved selecting two vertices (classes) and adding them in two separate partitions, one representing vertices that would remain on the adapting device and the other representing vertices that would be migrated to the target. All other vertices are initially added to the partition representing the target node, and are iteratively merged (moved) to the partition representing the adapting device, each time recording the utility of the new object-to-partition assignment (computed as a function of the four objectives discussed in section 2.3.2.2). Finally after all except the initial vertex in the target node partition have been merged to the partition of the adapting device, the recorded utility values are compared and the most highly scoring object-to-partition assignment implemented by the adaptation engine.

More recently, Shumao et al. (Ou et al., 2006) proposed a derived multi-level graph partitioning heuristic to adapt across multiple constrained devices by successively coarsening an application’s class graph. This was done by randomly selecting vertices and merging them with their lightest (low vertex weight) but highly coupled (high edge weight) neighbour, until the number of vertices was equivalent to the number of collaborating devices. Each vertex in the resultant coarse graph is then mapped to a device in the collaboration and represents a partition of the original graph. The coarsest resultant graph, which represents the partitioned topology, consists of only as many vertices as there are collaboration nodes, with each vertex or partition mapped to a single node within the collaboration (see Figure 2-2). This approach was shown to provide both better performance and efficacious adaptation (reduced edge-cut) in comparison to the approach by Gu et al. (Gu et al., 2003).

In both cases, constraints (thresholds) are placed on the size of each partition (subset), which influence the total weight and number of classes that can be grouped within it. These constraints typically correspond to either resource availability of a candidate remote node or the amount of resource mitigation required by the adapting device. The edge-cut in the partitioned class graph thus represents an estimation of the total network cost (due to remote procedure calls) that would occur in the new object topology as a result of the migration and is used as a measure of the quality of an adaptation.
As the process of partitioning generally aims to merge highly coupled classes, it ensures their co-locality thereby reducing inter-node communication of the resultant topology (Objective 4). Furthermore, the algorithm ensures that the weight of a vertex (or subset in the final graph), which is the cumulative resource usage of its constituents, obeys the constraints of its corresponding device thus avoiding over provisioning the candidate (Objective 3) and achieving the client’s criteria (Objective 2) as well as the application’s requirements (Objective 1).

2.3.3.3 Limitations

While unlike object-level approaches, class-level decision-making is able to address all objectives discussed in section 2.3.2.2, it has a number of efficiency and scalability limitations. Firstly, with an $O(|V|^3)$ running time complexity, where $|V|$ is the number of vertices, computing a graph partitioning decision is expensive and less scalable to application graph size. This cost is compounded in mobile environments, not only by the constraint of devices but by the frequency of decision making necessitated by execution in a dynamic environment. Secondly, the approach requires that each collaborating device maintain a copy of the application class graph. This incurs a collaboration-wide memory cost of $O(NM)$ where $N$ is the number of devices and $M$ is the application graph size (vertices and edges combined). In addition, since each device can only monitor the resource usage of its local components, it must rely on frequent updates of changes to remote components from other devices, which incur power, performance and network overheads on all devices. Lastly, each new adaptation decision ignores the current object topology, such that subsequent topologies may bear little resemblance to those that preceded them. Therefore, large migration costs (wherein many objects are migrated to many different hosts to represent the new topology) can potentially occur for only marginal gains in efficacy (i.e. reductions in edge-cut). Note that the migration cost of a single class consists of the transfer of its class file and all its serialised instances.
In addition to efficiency and scalability limitations discussed above, class-level adaptation approaches have limitations caused by the smaller number of classes relative to the number of objects, which is typical for computationally heavy applications. For example, during a simple run of a computationally heavy application (National Aeronautics and Space Administration, 2004) consisting of only 80 classes, over 12,000 objects are instantiated. Consequently, the class graph had relatively heavy vertices and edges as the large number of objects and inter object relationships were mapped to fewer class level vertices and edges. This inturn limits the flexibility of adaptation and the quality of object topologies generated as will be discussed in more detail in Chapter 4. In general, existing work for computing adaptive object mobility decisions (class-level and object-level) incurs costs associated with the process of selecting candidate devices and computing object topologies, which limits their feasibility in mobile and pervasive spaces.

Consequently, this thesis aims to address these limitations as follows: Firstly, Chapter 3 addresses the device selection processes (discussed in section 2.3.2.1); Secondly, Chapter 4 considers efficacy concerns of object topology computation (discussed in section 2.3.3.3); Thirdly, efficiency and scalability concerns (discussed in section 2.3.3.3) are addressed in Chapter 5. Finally, the approaches proposed in chapters 3-5 are methodically combined in Chapter 6 to create a new adaptation strategy for computing efficient, scalable and efficacious adaptive object topology decisions in resource dynamic mobile environments.
Chapter 3  Improving Efficiency and Scalability of Adaptive Candidate Device Selection

As discussed in Chapter 2 computing an adaptation decision proceeds in several steps the first of which involves the selection of candidate devices with which to adapt. This process enables a device to first identify available resources within a collaboration environment so that an optimal distribution of objects across devices can be computed (discussed in Chapters 4 and 5). However, existing implementations of this process involve underlying management overheads. Specifically, the current state-of-the-art approach for collaboration selection (Rossi P. and Ryan C., 2005) incurs overheads, which limit the efficiency and scalability of the adaptation process. Consequently, these overheads limit the utility of such approaches in resource constrained heterogeneous environments such as mobile and pervasive spaces.

Hence, in order to address these limitations a novel distributed approach to candidate device selection is proposed in this Chapter. The approach reduces the need to communicate information about collaborating devices and allows for the partial distribution of the adaptation decision-making process as discussed further in section 3.3. Through a combination of analytical modelling and simulation, it is shown that the network and memory utilization of the proposed algorithm is considerably smaller than existing approaches under various adaptation scenarios. Specifically, while for small collaborations, the existing algorithm offered up to 30% less network overhead, under medium to large-scale collaborations the proposed approach offered over 90% reduction in network consumption while maintaining linear memory complexity in contrast to the quadratic complexity of an existing approach. In addition, the proposed approach offered as much as a 96% reduction in power consumption.

This Chapter is organized into five subsections as follows; firstly, section 3.1 provides a discussion on existing candidate device selection strategies and identifies the current state-of-the-art approach upon which this thesis improves. Then, section 3.2 discusses and quantifies the efficiency and scalability limitations of this state-of-the-art approach with the aid of mathematical models. Section 3.3 proposes a novel distributed approach that addresses these limitations and reduces the overheads of
adaptation as shown through an evaluation in section 3.4. Lastly, a brief summary and outline of future work is provided in section 3.5.

3.1 Related Work

The candidate device selection process employed by an adaptation strategy is determined based on the direction of adaptation assumed. Specifically, adaptation could be performed one-way from a constrained mobile device to a resource copious server (section 3.1.1), or multi-way between multiple constrained devices in a peer-to-peer setting (section 3.1.2). These different modes of adaptation present different candidate device selection requirements as discussed below:

3.1.1 One-way Adaptation (Surrogate based Adaptation)

One-way adaptation techniques (Gu et al., 2003; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) generally assume the existence of a single mobile device and a dedicated unconstrained surrogate node that serves as an offloading target. Resource availability on the surrogate device is always assumed and adaptations are performed in a unidirectional manner in which only the mobile client device computes adaptation decisions, which offload (or on-load) its components. While some work (Ou et al., 2006; Ou et al., 2007b) has considered the possibility of surrogates with bounded resource capacities, adaptation still proceeds in a unidirectional manner in such approaches.

Typically, these approaches employ simple service discovery schemes such as UPnP (Jeronimo and Weast, 2003) and Jini (Arnold et al., 1999) in order to discover surrogates. In such approaches, a mobile client looks up available surrogates from one or more service registries. Once such a device is discovered adaptation offloading proceeds between the mobile-client and surrogate for the duration of the application execution. However, while this model is applicable to a simple scenario wherein only the resource availability within the mobile client device is dynamic, the scenario is unscaleable for adapting applications in mobile and pervasive environments wherein the presence of a dedicated resource copious device is not guaranteed and would also present a central point of failure.

3.1.2 Multi-way Adaptation (Peer-to-Peer Adaptation)

In Peer-to-peer adaptation (Gani, 2010; Gani and Ryan, 2009; Gani H. et al., 2006; Ou et al., 2007b; Philippsen and Zenger, 1997; Rossi P. and Ryan C., 2005;
Ryan C. and Rossi P., 2005; Ryan C. and Westhorpe C., 2004), the collaboration environment consists of a dynamic set of constrained devices, each of which could run out of resources and require load mitigation (adaptation) at some point during the collaboration. Unlike surrogate based approaches, multi-way adaptation does not require dedicated resource copious servers and is thus more applicable to mobile and pervasive spaces.

The most explicit multi-way adaptation approach is that by Rossi and Ryan (Rossi P. and Ryan C., 2005) in which the utility of placing each object to each device is evaluated by computing a score value based on the resource requirements of the object and resource availability of each device. This decision can be computed either by a dedicated central device (Global Adaptation) or by each collaborating device (Local Adaptation) (Rossi P. and Ryan C., 2005; Ryan C. and Rossi P., 2005).

In Global or Centralized Adaptation (Gani, 2010; Rossi P. and Ryan C., 2005; Ryan and Rossi, 2005; Ryan C. and Rossi P., 2005), a single dedicated machine is responsible for computing adaptation decisions. Other nodes within the collaboration periodically communicate their environment and application metrics (Gani H. et al., 2006) to this central node, with the metrics pushed out by a node, including the resource usage measurements of the device and the individual objects hosted by it. The central node is responsible for determining when an adaptation decision is required (e.g. a device runs out of resources etc.) and computing a new object to device distribution which optimizes the performance of the application and the load balance within the collaboration.

However, while the complete information maintained by the central device (about the collaboration environment and application behaviour), could allow for the computation of near optimal decisions, such decisions are shown to be computationally infeasible and unscaleable, as demonstrated in (Ryan and Rossi, 2005). Furthermore, the approach presents a central point of failure, which limits its applicability in mobile and pervasive spaces.

In Local or Decentralized Adaptation, decision-making is computed on individual nodes rather than on a single centralized device, in contrast to Global Adaptation. Resource metrics of each device are periodically communicated to every other node within the collaboration as shown in Figure 3-1. Unlike Global Adaptation, the metrics propagated include only the resource availability within the collaboration and not the software metrics of objects (Gani H. et al., 2006) (e.g. number of method invocations, method response times etc.). Figure 3-1 shows a simple collaboration environment and visualizes the metrics exchange from a client mobile device to the
rest of the collaboration and vice versa. Note that for simplicity Figure 3-1 omits the metrics exchange that occurs between every other device and the rest of the collaboration.

![Diagram of collaboration and metrics exchange](image)

**Figure 3-1 Local Adaptation Approach**

In the Local Adaptation approach proposed by Rossi and Ryan (Rossi P. and Ryan C., 2005), when a node runs out of resources (or detects changes in environmental factors that require an adaptation), it computes an adaptation decision based on the information it maintains about the collaboration and the metrics of the objects in its memory space using the algorithm outlined in Figure 3-2.

```
1  maxScore = 0.5
2  DO
3     maxObject = null, maxNode = null
4     FOR EACH mobile object o in local node DO
5         FOR each remote node n DO
6             score = evaluate(o, n)
7             IF (score > maxScore) THEN
8                 maxScore = score
9                 maxObject = o
10                maxNode = n
11             END IF
12         END FOR
13     END FOR
14     IF (maxScore > 0.5) THEN
15         move maxObject to maxNode
```

Problems

1. Efficiency
2. Scalability
CHAPTER 3 Improving Efficiency and Scalability of Adaptive Candidate Device Selection

Figure 3-2 Local Adaptation Algorithm basic flow, Rossi and Ryan (Rossi P. and Ryan C., 2005)

The evaluate() function (called in line 6) computes a score which quantifies the suitability of placing each mobile object, o on each remote target node, n. The score of an object placement (placement of object o to node n) considers two objectives: resource offloading from the constrained node, and performance improvement of the application. The resource offloading score evaluates the degree to which the load of the source device \( \frac{ru_c}{rc_c} \) can be mitigated and the load difference with the target node reduced (equation (1)) whereas the performance score evaluates the degree of response time improvement achievable through the migration of an object to a target (equation (2)). Note that these objectives serve as a basic example, which can be readily extended to include other objectives such as power conservation, reliability etc. The results of these two factors are aggregated using a weighted power mean as shown in Equation (3) after which the highest scoring object to device placement option is selected and implemented. The process is repeated until either all mobile-objects are migrated or an object-to-node match that can achieve a minimum threshold score is no longer available. Hence, the algorithm has a running time complexity of \( O(M^2N) \) where \( M \) is the number of mobile objects and \( N \) is the number of collaborating devices.

Since the adaptation decisions are computed by considering the behaviour of only a subset of the overall objects, the decisions made are not as optimal as the centralized approach (Global Adaptation). However, unlike Global Adaptation, Local Adaptation offers a more computationally efficient and scalable approach to adaptation decision-making and does not present a central point of failure.

\[
ru_d = \frac{ru_c}{rc_c} - ru_d\]

\[
rt_d = rt^{o}_{src} - rt^{o}_{dst} - MT
\]

\[
s = \left( W_{mu}f^{r}_{mu} + W_{nu}f^{r}_{nu} + W_{pu}f^{r}_{pu} + W_{ru}f^{r}_{ru} \right)^{1/r}
\]

Formula expressing the calculation of resource utilization differential score (1), the response time improvement score (2) and the aggregate score (3) computations of an object (Rossi P. and Ryan C., 2005)
3.2 Limitations of Existing Approaches

To estimate the resource usage of the approach proposed by Rossi and Ryan (Rossi P. and Ryan C., 2005), the author of this thesis derives models shown in equations (4)-(6). Equations (4) and (5) estimate the amount of memory consumed on each device and the entire collaboration respectively as a function of the number of devices $N$, the environment metrics $E$, and the software metrics propagated $S$. Similarly, Equation (6) shows the estimated network utilization cost of this approach by using additional parameters for the duration of application execution $T$ (seconds) and the frequency of metrics updates $F$, which is stated in terms of $\text{propagations second}$ and indicates the number of metrics updates (software and environmental) communicated within the collaboration environment per unit time. The inter-object invocation cost between every pair of objects $o$ and $p$ residing in different devices is symbolized in equation (6) as $o \rightarrow p$, and refers to the total network cost of object $o$ invoking methods of object $p$ (but not vice versa).

\[
mu_n = N \sum_{n \in N} \mu(e_n) + \sum_{o \in O_n} [\mu(o) + \mu(s_o)] \quad \text{where } e_n \in E, s_o \in S \tag{4}
\]

\[
mu_{total} = N^2 \sum_{n \in N} \mu(e_n) + \sum_{o \in O} [\mu(o) + \mu(s_o)] \tag{5}
\]

\[
nu_{total} = 2tf(N^2 - N) \left( \sum_{n \in N} nu(e_n) \right) + \sum_{o \in O} \left( nu_{ext}(o) + \sum_{p \in O \setminus o} nu(o \rightarrow p) \right) \tag{6}
\]

Based on these models it is observed that, in the context of mobile and pervasive spaces, the local adaptation approach defined by Rossi and Ryan (Rossi P. and Ryan C., 2005) (and outlined by the algorithm shown in Figure 3-2) presents three basic shortcomings as outlined below:

1. As resource availability is inherently dynamic, suboptimal decisions could be made based on out-of-date information about the collaboration. Therefore, avoiding this problem requires nodes to communicate their environment metrics
CHAPTER 3 Improving Efficiency and Scalability of Adaptive Candidate Device Selection

more frequently. However, with an \(O(N^2)\) message complexity\(^1\) for a single collaboration-wide communication (inferred from equation (6)), metrics updates are unscaleable to larger collaboration settings. In addition, the storage and maintenance of environmental metrics on each device requires additional memory resources with a quadratic order of space complexity \(O(N^2)\) as inferred from equation (5).

2. With a runtime complexity of \(O(NM^2)\), where \(M\) is the number of mobile objects and \(N\) is the number of collaborating devices, decision making is computationally expensive for mobile devices; exacerbated by the fact that the decisions are often performed by devices which are low on resources (i.e. in the case of devices adapting to mitigate their resource load). In response, nodes need to set lower constraint thresholds for triggering adaptation so that enough resources to compute adaptation decisions are reserved. This lowered threshold in turn results in increased adaptation throughout the collaboration causing additional network overheads and reduced application performance.

3. In addition, the heterogeneity of the environment incurs additional computational overheads to the decision making process. This is a result of the diversity of devices in pervasive environments wherein adaptation algorithms would need to consider additional metrics, such as location, power usage, reliability etc. This in turn requires adapting nodes to apply different score computation models for different devices based on the metrics relevant to each target node, thus introducing additional computation complexities to the adaptation logic.

While the constrained nature of the devices in pervasive environments requires that an adaptation algorithm compute an optimal decision with minimal computation resources, the heterogeneity and indeterminate size of the collaboration requires that the solution scale adequately to diverse collaboration environments and application behaviour. Thus, while the existing adaptive candidate device selection approach is applicable to small-scale homogenous collaboration settings it is unscaleable and costly in medium to large-scale heterogeneous collaboration environments as discussed further in the evaluation in section 3.4.

---

\(^1\) Message Complexity, in this context, is based on the number of messages sent and received by each node within the collaboration.
3.3 Distributed Adaptive Candidate Device Selection

Hence, this chapter focuses on improving the limitations of the local adaptation strategy identified in the previous subsection by proposing a distributed approach to candidate device selection. Underlying this new approach is a strategy of computing an adaptation decision, which reduces the need for metrics communication and disassociates the optimality score of a candidate device from its utility to individual objects. The proposed approach partially distributes the decision making process thus reducing the associated computation costs on the adapting node. Furthermore, the algorithm reduces the network and memory utilization costs that arise from increased collaboration size and heterogeneity.

The approach first involves each node connecting to a multicast address through which all adaptation related communication would take place. The use of multicast groups for communication allows nodes to delegate the responsibility of maintaining an awareness of the collaborating nodes to external network devices (e.g. routers, switches). Once connected, each node monitors its own environment metrics and the metrics of the objects within its memory space, however unlike existing approaches this information is not communicated to other nodes. The anatomy of the distributed approach is illustrated in Figure 3-3 and described as a five-step process below:

Figure 3-3 Distributed adaptation decision-making algorithm

```plaintext
1  FUNCTION adapt()
2     AdaptationRequest r = generateAdaptationRequest()
3     Object[] objects = getLocalObjects()
```
CHAPTER 3 Improving Efficiency and Scalability of Adaptive Candidate Device Selection

```java
4   Double timeout = calculateTimeOutFor(r)
5   multicastAdaptationRequest(r)
6   Device[] fittestCandidates = getFittestCandidateResponses()
7
8   // Discussed in Chapters 4-6
9   computeObjectTopology(fittestCandidates, objects)
10 END FUNCTION
11
12 FUNCTION getFittestCandidateResponses()
13   Device[] fittestCandidates;
14   WHILE !hasTimedOut()
15     // wait for a response or return if the timeout value
16     // expires
17     Device d = getResponse(timeout)
18    /*
19           Ensure that the device meets the requirements of the
20           adapting device
21        */
22     IF (isValidResponse(d))
23       fittestCandidates.addElement(d)
24   END WHILE
25 END FUNCTION
26
27 FUNCTION generateAdaptationRequest()
28   URI uri = getRuntimeURI()
29   MetricContainer envMetrics = getEnvironmentalMetrics()
30   MetricContainer ruMin = getMinObjectRU()
31   MetricContainer ruMax = getMaxObjectRU()
32   Double threshold = determineAdaptationThreshold(envMetrics)
33   RETURN new AdaptationRequest(uri, envMetrics, ruMin, ruMax, threshold)
34 END FUNCTION
```

Figure 3-4 Distributed Candidate Selections and Adaptation Algorithm

1. Adaptation Request

When a node is resource constrained, it multicasts an adaptation request, \( R \), containing its own environment metrics and synoptic information about the objects in its memory space (label 1. in Figure 3-3 and lines 2, 27-35 in Figure 3-4). The total message complexity in terms of bytes sent and received is \( O(N) \), since a single request is sent and \( N - 1 \) messages are received. The request sent by the node includes the following information, the purpose of which is explained in subsequent paragraphs and section 3.3.1:

- **Node URI**: This includes the IP address of the node and the port number of the middleware runtime.

- **Environmental Metrics (E)**: This includes the memory, processor, network utilization and capacity of the source node (adapting node).

- **Synoptic information about objects**: The least resource utilization consumed by an object \( ru_{O_{min}} \), and the total resource utilization consumed by all mobile
objects within the node, $ru^{0}_{\text{total}}$ are recorded for each metric type (memory, network, power, processor etc.).

- **Minimum Threshold Value, $k$:** The minimum score that can result in desirable migration (similar to the score threshold identified by Rossi and Ryan (Rossi P. and Ryan C., 2005)). This threshold is derived from the amount of constraint on the client device and the minimum amount of load mitigation or application performance improvement it aims to achieve through adaptation.

2. **Fitness Value Computation**

Each target node receives the adaptation request and determines its own suitability for adaptation by computing a fitness score. This fitness score represents the amount of resources the source can offload to the target, thereby reducing the load disparity between the two nodes; as well as the improvement in response time that the target can provide. Additionally, candidate nodes, which are already constrained, could choose not to compute adaptation decisions hence reducing the amount of global processor cycles consumed for an adaptation. The fitness value computation is further discussed in section 3.3.1.

3. **Delay Period Computation**

Each node compares its fitness score against the minimum threshold, $k$, sent by the source. If a node can offer a better score, it computes a delay period, which is inversely proportional to its score, thus providing fitter nodes with shorter delay periods. The node would then wait for the computed period before reporting its fitness score to the source. The benefit of this approach is two-fold: firstly, it reduces the possibility of a multicast sender-storm in which the source might be inundated by fitness responses; and secondly, it allows for the suppression of further fitness reports once the fittest candidate has responded (see 4. below), thereby reducing network costs. Though such a delay scheme would introduce latency in the decision making process, it occurs parallel to the execution of the application and hence is less likely to reduce application performance. As various delay period computation models exist and could be adopted for this scenario, in section 3.4 the approach is evaluated under different degrees of effectiveness of such delay schemes, and as such a specific delay scheme is not assumed in this work.

4. **Reporting the Fitness Score**

Once the delay period of a candidate expires it multicasts a response to the collaboration, which is received by the source and every other candidate within the
collaboration. Then each candidate compares the offered fitness score against its own and if the offered fitness value is greater, the candidate cancels its delay timer and need not reply to the source. However, if the target node can offer a better fitness value, it will continue to wait for its delay period to expire before communicating its fitness score in the same procedure, thus giving precedence to any other node, which could be fitter than it. In the event that multiple fit candidate responses are required, each node waits until either its time expires or a maximum number of fit candidates have responded before cancelling its fitness score. Enforcing the delay timer on every response, assures that the source receives responses from only the fittest candidate(s), hence minimizing resource costs and avoiding a sender-storm problem. The response that is multicast by a candidate includes the following information:

- **Node URI:** This includes the IP address of the candidate and the port number of its middleware runtime.

- **Environment Metrics (E):** The capacity and utilization of each resource metric type (memory, processor, network) of the target node.

- **Fitness Score, (S):** The fitness score computed by the target node, which is a value between 0 and 1 (Discussed in section 3.3.1).

- **Required Object Metrics Value, \( ru^O \):** The \( ru^O \) is a value between the \( ru^{O_{\text{min}}} \) and \( ru^{O_{\text{total}}} \) value sent by the source (step 1), which describes the resource utilization of a theoretical object cluster for which the specified fitness score can be achieved (discussed further in section 3.3.1). The source node will group objects so that the overall resource consumption of the group matches this specified value (discussed below).

5. **Clustering of Objects**

   The source node listens to fitness score multicasts for a pre-computed duration (lines 13-26 in Figure 3-4). This wait period is based on the amount of time it would take a node achieving the minimum threshold score \( k \) (discussed in step 1 above) to respond, accommodating for expected latency based on the bandwidth of the source device. If multiple fitness scores have been received by the time the wait period of the source device expires, the node selects either the best-offered score, if a single device is required, or the best set of devices, if a fixed number of adaptation targets are required. The source node then groups objects within its memory space to meet the criteria, \( ru^O \), required by the selected candidate. The identified object cluster would be a subset of the mobile objects within the memory space of the constrained source
node. Once the grouping of objects is complete, the source migrates the cluster to the candidate node. The migration of object clusters instead of individual objects reduces object spread, and decreases inter-object network communication cost, thereby improving the optimality of adaptive decision-making when compared to the existing approach; this aspect is discussed further in Chapters 4-6.

### 3.3.1 Fitness Score Computation

As discussed in section 3.3, the fitness score determines the degree of suitability of each candidate to the adaptation request, and guides the object grouping performed on the source node. Hence, to a certain extent, the fitness score computation model is dependent on the object topology computation strategy employed (as is discussed later in this subsection). This section proposes a fitness score computation model which is partly based on the approach proposed by (Rossi P. and Ryan C., 2005). The computation approach considers the resource-offloading objective from the constrained node for each metric type, which evaluates the degree to which the load of the source node can be mitigated and the load disparity with the target node reduced as discussed further in section 3.3.1.1 below. The objectives considered can be extended to include more diverse goals such as power preservation, reliability etc.

#### 3.3.1.1 Resource Offloading Score

While the existing approach computes a resource offloading score for individual objects, the distributed approach has the flexibility of attaining higher scores by grouping mobile objects together. The candidate computes an ideal $ru^O$ value, which is the resource utilization of a hypothetical object cluster for which it can offer the most ideal score for each metric type. Equation (7) shows the resource offloading score computation proposed by Rossi and Ryan (Rossi P. and Ryan C., 2005). The resource utilization difference, $ru_d$ computes the difference between the current and future load disparities between the two machines, wherein the future load difference refers to the load after the migration of object $O$ from the current node, $C$ to the destination node, $D$. This is computed for each metric type, (memory, network and processor) separately. Higher values indicate the migration would result in improved load balance between the two while offloading sufficient resources from the source.

The resource utilization difference, $ru_d$ is a linearly increasing function of $ru^O$ until the upper bound $ru_d^{max}$ (Equation (8)) is reached. This marks the point where, the hypothetical migration would result in 0 load difference between the two nodes. After this point $ru_d$ is a linearly decreasing function of the increasing $ru^O$. While the
existing work computes \( ru_d \) attainable for individual objects, the proposed approach has the flexibility of attaining higher \( ru_d \) values by grouping objects together.

Hence the computation performed by the new algorithm determines the best possible \( ru_d \) achievable for the objects on the source node by using the synoptic information of the minimum and total \( ru^O \) transmitted by the source (discussed in step 1 of section 3.3).

The best achievable \( ru_d \) is not necessarily the upper bound, but rather one of three possible values: 1) the \( ru_d \) upper bound achieved for resource utilization \( ru^O_{\text{max}} \), Equation (8), 2) the \( ru_d \) achieved by the sum of the resource utilization of the mobile objects (Equation (9)) or 3) the \( ru_d \) for the minimum object resource utilization (Equation (10)). This is because the source node might not have sufficient resources to achieve \( ru^O_{\text{max}} \) with the target. For instance, a target might achieve a memory utilization score of 0.6 if the source offloads 5MB worth of memory data, but the total sum of the memory utilization of all objects in the source could be less than that. The best achievable \( ru_d \) for each metric type is hence determined by selecting the maximum \( ru_d \), for which the source can offer a matching \( ru^O \) as is shown in Equation (11).

\[
ru_d = \frac{ru^C - ru^O}{rc^C - rc^O} - \frac{ru^C - ru^O + ru^O}{rc^C - rc^O} \quad \text{with} \quad ru^O = ru^F + \sum_{n=1}^{m} (ru^N_n/n)
\]

(7)

\[
ru^O_{\text{max}} = \frac{ru^C - ru^O}{rc^C - rc^O} \quad \text{when} \quad ru^O_{\text{max}} = \frac{rc^O ru^C - rc^C ru^O}{rc^O + rc^C}
\]

(8)

\[
ru^O_{\text{total}} = \frac{ru^C - ru^O_{\text{total}}}{rc^C - rc^O} \quad \text{with} \quad ru^O_{\text{total}} = \frac{ru^O + ru^O_{\text{total}}}{rc^O + rc^C}
\]

(9)

\[
ru^O_{\text{min}} = \frac{ru^C - ru^O_{\text{min}}}{rc^C - rc^O} \quad \text{with} \quad ru^O_{\text{min}} = \frac{ru^O + ru^O_{\text{min}}}{rc^O + rc^C}
\]

(10)

\[
\text{max} \{ ru^O \mid ru^O_{\text{min}} \leq ru^O \leq ru^O_{\text{total}} \} \cap \{ ru^O_{\text{max}}, ru^O_{\text{dmin}} \}
\]

(11)

3.3.1.2 Aggregate Score Computation

While the resource offloading score is computed for each individual metric type, an aggregate score that considers the individual scores of each metric type needs to be computed to evaluate the overall suitability of a node. Rossi and Ryan (Rossi P. and Ryan C., 2005) propose both linear and non-linear approaches to aggregate score computation (Equation (12) & Equation (13)). The linear approach (Equation (12)) allows for a weight based favouring of metrics, whereas the non-linear approach provides means of indicating neutrality of the function. The distributed approach
adopts both these methods for aggregate computation. However factoring in the $ru_d$, computed for each metric type in the earlier section, to the aggregate score computation would be inaccurate. This is because a cluster that meets the resource utilization score of one metric type does not necessarily guarantee the satisfaction of the $ru_d$ of the other.

For instance, a target might compute a memory utilization offloading score of 0.6, a network utilization score of 0.5, and a processor utilization score of 0.4, the cluster achieving any one of these metric scores might not satisfy the others. Hence, the new algorithm computes the average aggregate score for each metric type by considering two cases (Equation (16)): 1) When all other metric types achieve their worst possible score within the range of objects specified by the source (equation (14)), and 2) When all other metric types achieve their best score for the same range of objects (Equation (15)). The worst case for each metric type is computed in a similar way to Equation (11), but by selecting the minimum value instead of the maximum of the three computed $ru_d$s. Equation (14)-(16) show the use of the linear method proposed by the Rossi and Ryan (Rossi P. and Ryan C., 2005) to compute an average score value for metric $I_1$ (where I is a normalized value of $ru_d$) (Rossi P. and Ryan C., 2005) in which $I_1$’s best score is used. The average scores are computed for each metric type and the greatest of the three selected as a final score. As such, this $S$ value is an estimate of the achievable score when clustering is focused on the most ideal metric type on that target. This allows for a single objective clustering on the source, instead of clustering based on multiple criteria (clustering to meet memory, network and processor needs at the same time), which is computationally expensive.

$$S = W_{mu}l_{mu} + W_{nu}l_{nu} + W_{pu}l_{pu}$$  \hspace{1cm} (12)

$$S(r) = \left( W_{mu}l_{mu} + W_{nu}l_{nu} + W_{pu}l_{pu} \right)^{1/2}$$  \hspace{1cm} (13)

$$S_{worst}^{I_1} = W_{1}l_{1}^{best} + W_{2}l_{2}^{worst} + W_{3}l_{3}^{worst}$$  \hspace{1cm} (14)

$$S_{best}^{I_1} = W_{1}l_{1}^{best} + W_{2}l_{2}^{best} + W_{3}l_{3}^{best}$$  \hspace{1cm} (15)

$$S_{average}^{I_1} = \frac{(S_{best}^{I_1} + S_{worst}^{I_1})}{2}$$  \hspace{1cm} (16)

However, it is noted that since the applicability of a given fitness score model is dependent on the specific object topology computation strategy employed, the model proposed in this section is later superseded (in Chapter 6) by a model that accounts for a more efficacious object topology computation strategy than that used by (Rossi P. and Ryan C., 2005) as discussed in more detail in Chapters 4-6.
3.3.2 Analysis of the proposed approach

By allowing candidate nodes to compute their own fitness value, the approach removes the need to periodically communicate environmental metrics. This in turn reduces network communication costs and other resource overheads associated with storing and maintaining collaboration information on every node. Consequently, the resource utilization of the distributed approach is more scalable to the increase in collaboration size compared to the existing approach as evaluated in section 3.4. Furthermore, by dividing and offloading the tasks of the adaptation process, the space and time complexity of decision-making is reduced, making adaptation more feasible and efficient for the local constrained device.

Equation (17) below shows that the memory consumed on each device by the distributed approach, is the sum of the memory utilization of: 1) the environmental metrics $E$ of the device; 2) its runtime objects $O_n$, and their 3) associated software metrics, $S$ (object memory usage, processor utilization metrics etc.). Consequently, the collaboration-wide memory consumed by the distributed algorithm is expressed in equation (18) below as the sum of the memory utilization of 1) the environmental metrics stored on each device 2) the software metrics stored on each device and 3) the overall memory utilization of the application. These models exclude the memory of external factors such as the middleware framework on which the adaptation system is implemented. It is seen from equation (18) that the proposed approach maintains linear memory complexity in contrast to the quadratic complexity of the existing approach (equation (5) in section 3.2)

$$mu_n = mu(e_n) + \sum_{o \in O_n} [mu(o) + mu(s_o)]$$  \hspace{1cm} (17)

$$mu_{total} = N \sum_{n \in N} mu_n$$  \hspace{1cm} (18)

Similarly, the estimated network utilization of the distributed algorithm in terms of the number of bytes sent and received by devices within the collaboration is expressed through equation (19) below by taking into account the number of Adaptations $Na$ that occur throughout the collaboration duration, the set of candidates that responded to an adaptation source during each adaptation, $C$ and the size of fitness message report $M$ that they send. Specifically it is shown that the network utilization of the proposed approach is the byte sum of: 1) each adaptation request $R$; 2) each message report $M$; 3) the total remote method invocations between
every pair of objects $o$ and $p$ residing in different devices, which is symbolized as $o \rightarrow p$, which refers to the total cost of object $o$ invoking methods of object $p$ (but not vice versa); 4) the migration of objects during every adaptation and, 5) any external network utilizations of the application (e.g. http request etc.)(\(nu_{ext}\)). The approach is seen to incur linear network utilization complexity in comparison to the quadratic complexity of the existing approach (equation (6) in section 3.2). In addition, since adaptation decisions are made using live metrics as opposed to cached metrics, the possibility of making suboptimal adaptation based on out-of-date metrics is minimized.

The delegation of score computation to the remote nodes also allows individual nodes to easily factor in a diverse range of additional metrics into the decision computation process without incurring further overheads on the collaboration environment, thus making the approach scalable to collaboration diversity and consequently more applicable to heterogeneous environments in comparison to the existing approach.

Moreover, the approach reduces resource contention between the executing application and the adaptation engine. Hence, unlike the existing approach, in which low constraint thresholds must be set to reserve resources for adaptation (section 3.2), applications are able to execute until higher constraint thresholds are reached. Consequently, this higher threshold reduces the number of adaptations and adaptation related overheads within the collaboration.

While it is worth noting that the distributed approach could result in longer adaptation decision-making times as a result of network latency and the delay scheme discussed in section 3.3, this is not expected to have a direct effect on the performance of the application because adaptation decisions are performed in parallel to the executing application. Furthermore, it is expected that for large-scale collaborations adapting computationally heavy applications, the execution of adaptation decisions on a local constrained device using the existing approach might take longer than the distributed approach, which leverages externally available resources.
3.4 Evaluation

In this section a comparative evaluation of both algorithms with respect to their memory, network and power utilizations is provided under varying environmental scenarios, by utilizing the mathematical models discussed in the sections 3.2 and 3.3. Both the maximum degree with which one algorithm outperforms the other, as well as the specific environmental scenarios for which each algorithm provides comparatively lower resource utilization, are identified and discussed in detail in sections 3.4.2-3.4.4. Since the efficacy of a given fitness score model is dependent on the specific object topology computation strategy employed, this aspect of the proposed approach is evaluated in Chapter 6 in the context of a more efficacious object topology computation algorithm than that used by (Rossi P. and Ryan C., 2005). Similarly, a valid comparison of the performance of the collaboration management approaches requires a consideration of the performance of the object topology computation process employed, as these processes are tightly coupled in the existing approach as discussed in section 3.1.2. Hence, the performance aspect of the candidate device selection approaches is also evaluated in Chapter 6, in the context of a state-of-the-art object topology computation approach. Evaluations of both aspects in Chapter 6 show that the approach offers both improved performance and better efficacy of adaptation decisions computed.

To this end, this section is organized into four subsections as follows: section 3.4.1 first provides a description of the environment variables affecting both algorithms, which are varied in order to simulate different environmental conditions. Then, sections 3.4.2 and 3.4.3 compare the network and memory utilizations of the proposed algorithm relative to the existing approach, under different collaboration and applications sizes. Finally, section 3.4.4 evaluates the power consumption of each approach under a small-scale adaptation setting involving three mobile devices.

3.4.1 Environmental Settings

In order to compare the resource utilization of both algorithms under diverse environmental settings, a range of possible values for the variables identified in equation (4)-(6) (existing approach) and equations (17)-(19) (distributed approach) are identified and discussed in this subsection. To evaluate the resource utilization of the algorithms independently from their adaptation outcome, the adaptive decisions made by both algorithms are assumed to be the same. However, it is expected that the adaptation decisions computed by the proposed approach would be more efficacious
for two primary reasons: 1) Adaptation decisions are computed on live metrics instead of cached metrics which is the case in the approach by (Rossi P. and Ryan C., 2005) and 2) decisions are computed on object clusters which allows matching of resources based on ideal load mitigation requirements as discussed in section 3.3.1. Moreover, this approach to decision computation allows for factoring of inter-object coupling information which improves efficacy of adaptation decisions as discussed in more detail in Chapters 4 and 5. This is in fact shown to be the case in the evaluations in Chapter 6, wherein the proposed approach is shown to yield more efficacious adaptation decisions.

Table 3-1: Variables influencing network utilization and memory utilization in both algorithms

<table>
<thead>
<tr>
<th>Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes (N)</td>
<td>(4 \leq N \leq 50)</td>
</tr>
<tr>
<td>Execution Time (T)</td>
<td>(600s \leq T \leq 86400s)</td>
</tr>
<tr>
<td>Frequency of propagation (F)</td>
<td>(0.0017/s \leq F \leq 0.1/s)</td>
</tr>
<tr>
<td>Number of adaptations (Na)</td>
<td>(Na \leq Np)</td>
</tr>
<tr>
<td>Number of propagations (Np)</td>
<td>(1 \leq Np \leq 8640), (Np = ET \times F)</td>
</tr>
<tr>
<td>Number of fitness values (Nf)</td>
<td>(Nf \leq N - 1)</td>
</tr>
<tr>
<td>Fitness report message size (M)</td>
<td>(M = 2E)</td>
</tr>
<tr>
<td>Environment metrics size (E)</td>
<td>(200\text{Bytes} \leq E \leq 10000\text{Bytes})</td>
</tr>
<tr>
<td>Adaptation request size (R)</td>
<td>(R = 2.5E)</td>
</tr>
</tbody>
</table>

- **Number of Nodes (N):** In order to observe the scalability of both algorithms with regards to collaboration size, a range of possible collaboration sizes ranging from a small-scale collaboration of 4 nodes, to a large-scale collaboration of 50 nodes, are considered.

- **Execution Time (T):** The overall time spent executing and adapting a given application for the purpose of this simulation ranges from a brief collaboration of 10 minutes up to 24 hours.

- **Frequency of Metrics Propagation (F):** This variable is applicable to the existing algorithm and refers to the frequency of propagating environmental metrics throughout the collaboration. A periodic metrics propagation is assumed for simplicity, though node triggered propagations based on degree of metrics change could be used (Gani H. et al., 2006). The greater the fluctuation in resource availability within an environment the more frequently metrics propagation needs to occur. The higher value of 0.1/s (propagation
every 10 seconds) would be more applicable to dynamic heterogeneous environments.

- **Number of Metrics Propagations** ($N_p = T \times f$): The number of times a collaboration wide metrics communication occurs in the *existing* algorithm is the product of the *Execution Time* ($T$) and the propagation *frequency* ($F$). High number of propagations means either long executing applications, high frequency of metrics propagation or both. Hence, a high number of propagations would be expected when executing long running applications in dynamic environments.

- **Number of Adaptations** ($N_a$): The number of adaptations cannot exceed the total number of metrics propagations that occur within a given period. This is because the event in which a node exceeds its resource utilization constraint thresholds, hence requiring an adaptation, would be a less frequent occurrence than a metrics propagation which reports resource changes of much lesser degree. The reason behind this theoretical upper bound is that for an adaptation to occur in the existing algorithm, the adapting node would need to first have information about the collaboration. This implies that at least one collaboration-wide metrics propagation needs to occur prior to any adaptation. As every adaptation would cause notable resource utilization differences within the environment, it needs to be followed by a subsequent propagation. This means that for every adaptation there would be at least one additional corresponding propagation. As the adaptation decisions and number of adaptations performed by both algorithms are assumed to be the same in this evaluation, it follows that the upper bound of $N_a$ for both algorithms would be $N_p$.

- **Number of Fitness Reports Returned** ($N_f$): While the number of fitness values returned for each adaptation request could vary, for simplicity we assume a fixed $N_f$ value in which a specified number of nodes reply to every adaptation request. We consider a theoretical upper bound in which $N_f$ is equal to the number of candidates, for a situation in which no functional delay scheme exists. However it is recalled that the delay scheme, discussed in section 2, would significantly reduce the number of fitness responses by allowing only the fittest nodes to respond. Hence, this assumption represents the worst-case scenario of the proposed candidate device selection approach.
- **Fitness Report Message Size** \(M\): As discussed in section 3.3, the report communicated to candidates consists of more information than the simple environment metrics, \(E\), propagated by the *existing* algorithm. Specifically, \(M\) is two times the message size of the *existing* algorithm.

- **Adaptation Request Message** \(R\): Similarly, it is determined that the adaptation request is 2.5 times the environmental metrics message size of the *existing* algorithm. This is established by measuring the serialized size of a simple adaptation request message containing the information discussed in section 3.3, encapsulated as a java object.

### 3.4.2 Network Utilization

Network utilization is assumed to be the total number of bytes sent and received by all nodes within the collaboration. Hence, a unicast of an environment metrics message, \(E\), from one node to the entire collaboration would cost \(2(N - 1)E\) whereas a multicast of the same information would cost \(N \times E\).

Equation (3) in section 3.2 modelled the total network utilization consumed by the existing algorithm of Rossi and Ryan (Rossi P. and Ryan C., 2005) during a complete collaboration session to be the sum of: 1) the total bytes of all metrics propagations; 2) the network utilization of each object due to *distributed* placement; 3) the cost of migrating objects after adaptation and 4) any external network utilizations of the application (*e.g.* http request etc.). On the other hand, the network utilization of the *distributed* algorithm was given in equation (19) in section 3.3 as the byte sum of: 1) the adaptation requests sent; 2) the adaptation reports responded for each adaptation; 3) the inter-device communication resulting from the placement of coupled objects on different devices; 4) the cost of migrating objects after adaptation; and 5) any external network utilizations of the application (*e.g.* http request etc.)(\(NU_{ea}\)).

A constant upper bound for the number of fitness values returned is first assumed \((N_f = N - 1)\) in order to visualize the network utilization of each algorithm under varying numbers of metrics propagations \((N_p)\), adaptations \((N_a)\), and collaboration sizes \((N)\). Setting \(N_f\) to its upper bound for each adaptation, models the scenario in which no functional delay scheme exists in the new algorithm thereby providing a worst-case comparison of the network utilization of the *distributed* algorithm versus the best-case scenario of the *existing* approach.
Figure 3-5 The environment scenario for which the \textit{existing} approach provides better network utilization. $\text{nu}(\text{existing}) < \text{nu}(\text{distributed})$

Figure 3-6 The environment scenario for which the \textit{distributed} approach provides better network utilization. $\text{nu}(\text{distributed}) < \text{nu}(\text{existing})$

Figure 3-5 & Figure 3-6 show a 3D region of the values of $Na$, $Np$ and $N$ for which the network utilization of the \textit{existing} algorithm would be better than the \textit{distributed} approach and vice-versa. The region is determined based on a simple 3D \texttt{regionplot} performed using the computational tool Mathematica (Wolfram Research, 2009) for the inequality $\text{nu}(\text{existing}) < \text{nu}(\text{distributed})$ and vice-versa as per equations (6) and (19) under the constraints specified in section 3.4.1. Note that the costs incurred by external application network utilization, object placement and object migration cancel out as they are assumed to be the same for both algorithms.
Figure 3-5 shows that the *existing* algorithm provides comparatively better network utilization in small collaboration sizes and very high numbers of adaptations ($Na \geq 65\%Np$). However, in larger collaboration sizes, the *existing* algorithm provides better performance only at increasingly higher numbers of adaptations where $Na \cong Np$. Computing for the maximum disparity between the two algorithms within this region, shows that the *existing* algorithm could offer as much as 35% less network utilization when $Na = Np$ (the theoretical upper bound for $Na$ discussed in section 3.4.1). In practice, a scenario in which the number of adaptations within the collaboration is as high as that favouring the *existing* algorithm is unlikely even in extremely dynamic computing environments. Such a high adaptation count suggests an object topology in a constant state of flux, thus implying that adaptation decisions are performed without benefits being gained, whilst adaptation overheads are still incurred. Furthermore, an event in which a node exceeds its resource utilization threshold, thus requiring an adaptation, would be a far less frequent occurrence than metrics propagation occurring for lesser degrees of change in resources.

Figure 3-6 shows the complementary region, under which the *distributed* algorithm provides better resource utilization. The figure shows that the proposed algorithm provides better network utilization under a greater portion of the environment settings, including under low number of adaptations in small collaborations and in all scenarios involving medium to large-scale collaborations. Computing the maximum degree of network utilization difference between the two algorithms under this region shows that the proposed *distributed* approach can provide over 90% reduction in network utilization under relatively low number of adaptations ($Na<10\%Np$) even in the absence of a delay scheme.

Figure 3-7 and Table 3-2 explore the effect of introducing a delay scheme with varying degrees of effectiveness into the proposed approach. Table 3-2 shows that the maximum degree with which the existing approach can outperform the distributed approach is considerably reduced with the introduction of a delay scheme with even a minimal degree of *effectiveness* (with *effectiveness* measured as the number of suboptimal fitness reports a delay scheme is able to supress). This can also be visualized as the shrinking of the region identified by Figure 3-5 and the expansion of the region shown in Figure 3-6. In addition, it is shown in Table 3-2 that with a delay scheme with more than 55% effectiveness (i.e. only 45% of the collaborating nodes reply on each adaptation) the *distributed* algorithm always outperforms the *existing* approach under all environment conditions (all values of $N$, $Na$ and $Np$).
3.4.3 Memory Utilization

Memory consumption, as modelled by the associated equations in sections 3.2 and 3.3, is defined as the number of bytes stored by each algorithm for the duration of the collaboration. This excludes temporary memory resident data such as adaptation request information from other nodes, or storage of serialized environmental metrics before propagation. The assumption also disregards memory utilization of the middleware as it would not bear upon the comparison of the individual algorithms.

The global memory consumed in the collaboration by the existing algorithm of Rossi and Ryan (Rossi P. and Ryan C., 2005) shown in Equation (5) in section 3.2 is the sum of the memory consumed by the following three factors: 1) environment
metrics of the entire collaboration stored on every node; 2) the metrics information stored about each object; and 3) memory utilization of each object. On the other hand, the total memory consumed by the \textit{distributed} algorithm, as shown in Equation (18) in section 3.3 is the sum of: 1) the environmental metrics of each node; 2) the metrics information stored about each object; and 3) the memory utilization of each object.

Hence a comparative look at equations (5) and (18) shows that while the \textit{existing} algorithm has a global memory complexity of $O(N^2)$ for storing collaboration information on every node, the \textit{distributed} approach has a more favourable memory utilization complexity of $O(N)$. Figure 3-8 shows that the difference between the memory utilization of both algorithms increases quadratically with the increase in number of nodes and linearly with the increase in environmental metrics size. The \textit{distributed} algorithm hence provides increasingly better (lower) memory utilization with an increase in the heterogeneity and size of the collaboration, thus making it more scalable to these environmental factors in comparison to the existing approach.

![Figure 3-8 Memory overhead saved by the distributed algorithm compared to the existing algorithm](image)

### 3.4.4 Power Utilization

At a minimum, the power consumption of each algorithm is a factor of four adaptation sub-processes 1) Collaboration management and device selection 2) Object topology computation 3) Object migration and 4) Inter-device network communication, resulting from the object topology computed in step 2). Since the focus of this Chapter is on candidate device selection subprocess, only the power
utilization of the first factor is evaluated in this section, whereas factors 2-4 are evaluated in Chapters 4-5.

To this end, an evaluation consisting of a small real world collaboration was used in preference to simulated test cases (e.g. simulated nodes and networks in OMNET++(Varga A., 2001)). Specifically the following three devices were used: 1) An HTC HD2 smartphone with a 1GHz Processor, 448 MB RAM running Android 2.3.7 2) An HTC Desire HD smartphone with 1GHz Processor, 768 MB RAM running Android 2.2 3) A MacBook Pro with 2.3GHz Intel Core i5, 4GB RAM running OS X 10.6.8. The devices were setup under laboratory conditions with all non-essential services stopped and all three devices connected through an IEEE 802.11g router.

A simple adaptation scenario was simulated, which involved all devices running a distributed adaptive application for a duration of 1 hour during which a number of collaboration-wide metrics communication and adaptations occurred. Specifically, a collaboration-wide metrics communication frequency of 0.5/minute (once every two minutes) was used for the existing algorithms, and an adaptation frequency of 15% of the number of metrics propagations was enforced. The values were selected to simulate a relatively stable collaboration environment in which adaptation decisions sufficiently account for the resource dynamism within the environment.

In the case of the existing algorithm, the process of ranking and selecting candidate devices (which is tightly coupled with the object topology computation process as discussed section 3.1.2) is omitted. This is to exclude the power cost of the object topology computation process from the power cost of the collaboration management, which is the focus of this section. This in turn means that the results obtained in Figure 3-9 underestimate the total power cost of the collaboration management process of the existing algorithm and thus understates the power utilization benefits of the proposed approach. In addition, in the case of the proposed solution, a scenario wherein no delay scheme exists is used throughout the evaluation (i.e. all nodes respond with fitness values to an adaptation request), thus simulating a worst-case scenario of the proposed solution in that regard.
Figure 3-9 Power utilization comparison of proposed algorithm vs. existing algorithm

It is shown in Figure 3-9 that the power utilization of the distributed algorithm is significantly less than the existing algorithm, with a 94.6% reduction in power consumption on the HTC HD2 and over 95.5% reduction in power consumption on the Desire HD in relation to the existing algorithm. This is inline with expected results, with the power utilization of the existing approach largely resulting from the need to communicate environmental metrics throughout the collaboration environment as discussed in section 3.1.2.

3.5 Summary

In order to address research question A) outlined in section 1.3.2, this Chapter focused on improving the efficiency and scalability aspects of the candidate device selection sub-processes of adaptation decision computation. It was shown that existing approaches incur a number of overheads stemming from two primary factors: 1) the computational cost of determining the efficacy of candidate devices and 2) the need to store, maintain and update collaboration-wide environment metrics on each device.

Hence, a distributed approach to local adaptation was proposed (section 3.3) to improve the efficiency and scalability of existing approaches. Through a combination of mathematical modelling and simulation, it was shown that, under a range of environmental conditions, the proposed approach offered an order of magnitude reduction in memory, network and power utilization costs associated with collaboration management (sections 3.4.2-3.4.4). In addition, it was shown that the approach was more scalable with regards to the size of the adapting application (in terms of number of objects and classes), the number of collaborating devices, the
resource availability fluctuation within an environment and the level of heterogeneity (in terms of the size and number of metrics maintained). Hence, the proposed approach addresses both the efficiency and scalability concerns of collaboration management and candidate device selection processes as outlined in research question A) in section 1.3.2.
Chapter 4  Improving Object Topology Computation Efficacy

As discussed in Chapters 1 and 2, the computational feasibility of an adaptation decision is largely determined by the optimality of the placement of objects across devices (adaptation efficacy). This aspect of adaptation influences both the efficiency of the adapting application and the computational cost underlying its distribution. However, existing approaches for computing adaptation decisions incur limitations as a result of the granularity at which adaptation decisions are performed. Specifically, while approaches that compute adaptation at a fine-granularity offer increased flexibility (i.e. offer more object placement possibilities) and potentially more efficacious decisions, they are computationally costly and unscaleable to heavy applications and large collaboration scenarios, as briefly discussed in Chapter 2. On the other hand, coarse-grained strategies offer improved efficiency in comparison to fine-grained approaches but limit adaptation flexibility thus resulting in potentially low efficacy object topologies.

Hence, this Chapter focuses on improving the efficacy of object topology computation by focusing on the granularity at which adaptation decisions are performed. To this end, a new type of granularity, which combines the efficacy of fine-grained adaptation with the computational efficiency of coarse-level strategies, is proposed. In addition, a novel approach for achieving this level of granularity through the dynamic decomposition of runtime class graphs is presented and empirically evaluated on a corpus of real world applications. It is shown that the approach improves the efficacy of adaptation decisions by reducing network overheads by a minimum of 17% to as much 99%, while maintaining comparable decision making efficiency to class level adaptation. Moreover, these improvements in efficacy translated into reduced power consumption and invocation latency in comparison to class-level approaches.

The chapter is structured into seven subsections as follows: First, section 4.1 provides a concrete definition of efficacy as a quality attribute of adaptation and then rationalizes the importance of optimizing this attribute in resource-constrained environments, by considering a hypothetical adaptation scenario. Next, section 4.2 discusses and analyses existing state-of-the-art approaches and their limitations with regards to adaptation efficacy, which is used in section 4.3 to form deductive
hypotheses about the impact of adaptation granularity on the efficacy of generated object topologies. These hypotheses are used as the basis for the proposed solution, which is discussed in detail in section 4.4 and evaluated in section 4.5. Then, section 4.6 provides a discussion of additional optimization approaches that can be performed to improve the efficiency of the proposed approach. Finally, section 4.7 provides a brief summary of the contributions presented in this Chapter from the context of the objectives of this thesis (as outlined in Chapter 1).

4.1 Motivation

An adaptation decision is typically computed either when a device within the collaboration is constrained or when the existing object-to-device placement is determined to be inefficient. The computed adaptation decision, which is described by the resulting object topology (object placement across devices), must optimize the resource consumption of the distributed application within the constraints (resource availability) of the collaborating devices. Specifically, as discussed in section 2.3.3, an adaptation decision must achieve the following four objectives:

**Objective 1: Meet the resource requirements of all objects:** A valid adaptation decision must provision application objects with the resources they require for computation (and the resources they require to meet Objective 2). For instance, an object that requires 1MB in memory can only be migrated to a target device that can at a minimum offer the same amount of free memory. In practice, various other metrics (CPU, power consumption etc.) are considered simultaneously when achieving this objective.

**Objective 2: Meet the resource requirements of the adapting client device:** A client device which computes an adaptation decision will have adaptation goals which mitigate its own load or improve the overall topology within the collaboration. For instance, these objectives could include reducing its memory load by 10% or improving application performance by 30% etc. These objectives are expressed differently by different algorithms as will later be discussed in section 4.2.

**Objective 3: Limit migrations to the resource provisions offered by the remote candidate:** Each collaborating device offers a specified amount of resources to the adapting client device. This amount expresses the resources that the device is willing to dedicate to the client device for that adaptation. Hence, an adaptation must ensure that devices are not overloaded, by limiting object to device assignments to the resource provisions offered by each device.
Objective 4: Minimize the total inter-device network costs that would result from placing objects apart: An inevitable overhead that results from the distribution of application objects is the inter-device communication resulting from remote method invocations. This cost impacts the entire collaboration through increased network consumption, which also translates into method invocation latency and power consumption.

Given the above four objectives, the validity and efficacy of an adaptation decision are consequently defined as follows:

Definition-1 Validity: An adaptation decision is said to be valid, if and only if the object topology computed satisfies the resource requirements of an application (objective 1) and the resource constraints of all collaborating devices (objectives 2 and 3 above).

Definition-2 Efficacy: A valid adaptation decision is efficacious if it minimizes the total inter-device communication cost of the object topology (objective 4). Hence, an object topology $t_1$ is said to be more efficacious than an alternative topology $t_2$, if the resulting inter-device communication cost of $t_1$ as expressed by objective-4 is less than that of $t_2$, provided that both $t_1$ and $t_2$ are valid topologies.

Hence, the objectives of an adaptation decision, which are expressed in terms of the degree of load mitigation required by a device or the performance optimization of an application (objectives 1-3), determine the validity of an adaptation decision. However, various object configurations might satisfy the above objectives (as will be illustrated through a simple example in proceeding paragraphs), and hence an efficacious topology is one that minimizes the unintended cost of object distribution, inter-device communication. This Chapter focuses on optimizing this aspect of adaptation decisions computed.

Table 4-1 Software Metrics of objects $O_1$, $O_2$ and $O_3$

<table>
<thead>
<tr>
<th>Objects</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>50KB</td>
</tr>
<tr>
<td>$O_2$</td>
<td>100KB</td>
</tr>
<tr>
<td>$O_3$</td>
<td>100KB</td>
</tr>
</tbody>
</table>

Table 4-2 Coupling amongst objects $O_1$, $O_2$ and $O_3$

<table>
<thead>
<tr>
<th>Inter-Object Coupling</th>
<th>Coupling Intensity$^2$</th>
</tr>
</thead>
</table>

$^2$ Coupling Intensity is quantified as the size of method parameters passed during method invocations per unit time. This is discussed in greater detail in sections 0 and 4.4
CHAPTER 4 Improving Object Topology Computation Efficacy

<table>
<thead>
<tr>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1 \leftrightarrow O_2$</td>
</tr>
<tr>
<td>$O_2 \leftrightarrow O_3$</td>
</tr>
<tr>
<td>$O_1 \leftrightarrow O_3$</td>
</tr>
</tbody>
</table>

Table 4-3: Alternative Configurations for simple applications between two devices

<table>
<thead>
<tr>
<th>Object Topology</th>
<th>Device 1 Objects</th>
<th>Device 2 Objects</th>
<th>Objective 1-3</th>
<th>Objective 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>{01, 02, 03}</td>
<td>{}</td>
<td>NO</td>
<td>N/A</td>
</tr>
<tr>
<td>$T_1$</td>
<td>{01, 03}</td>
<td>{02}</td>
<td>YES</td>
<td>1.2KB/Second</td>
</tr>
<tr>
<td>$T_2$</td>
<td>{01, 02}</td>
<td>{03}</td>
<td>YES</td>
<td>8KB/Second</td>
</tr>
<tr>
<td>$T_3$</td>
<td>{01}</td>
<td>{02, 03}</td>
<td>NO</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Improving adaptation efficacy is of importance in resource constrained heterogeneous environments because it directly translates into the network, power and the performance costs of the object topology computed. This can be demonstrated using an example adaptation scenario involving three objects $O_1$, $O_2$ and $O_3$, adapting across two devices, $D_1$ and $D_2$. Assuming that all objects initially execute on the client device $D_1$, which runs out of memory and offloads objects (adapts) to $D_2$ (with a load mitigation objective of 100KB of memory load reduction), it can be inferred using the simple but adequate metrics provided in Table 4-1 and Table 4-2, that either of the alternative topologies $T_1$ or $T_2$ shown in Table 4-3, can satisfy objectives 1-3 stated above. That is, migrating either object $O_2$ or $O_3$ but not both (topology $T_3$) to device $D_2$ would satisfy the requirements of all objects ($O_1$, $O_2$ and $O_3$) (objective 1), achieve the load mitigation requirements of the adapting device $D_1$ (objective 2) while not overloading the target device $D_2$ (objective 3), which offers only 100KB of memory. Hence, both $T_1$ and $T_2$ are valid adaptation decisions as per Definition-1 above.

However, it is observed that topology $T_2$, which involves migrating $O_3$ to device $D_2$, would incur an inter-device communication cost of 8KBytes/Second as a result of the coupling of $O_3$ with $O_1$ and $O_2$, in contrast to topology $T_1$ which would only incur 1.2KBytes/Second as a result of the migration of $O_2$ to device $D_2$. This indicates that effecting topology $T_2$ over $T_1$ would incur a 6.7-fold increase in network cost per unit time. In addition, the magnitude of this difference between the qualities of the two object topologies is highlighted further when considering its cumulative effect throughout the duration of the application’s runtime. Assuming an application execution duration of 30 minutes the network cost of $T_2$ grows to 14.4MBytes in in contrast to 2.16MBytes for topology $T_1$. Moreover, this cumulative effect of network
cost directly translates into proportionally increased power utilization and remote invocation latency (Discussed in section 4.5).

While the preceding scenario is not based on a real application and is simple, it highlights the importance of computing efficacious object topology decisions. In practice, object topology decisions often involve optimizing the placement of thousands of objects over a number of devices by factoring in their diverse resource requirements (memory, CPU, power etc.) and complex interaction patterns. This makes achieving an optimal solution computationally intractable and hence infeasible to compute (as will be discussed in more detail in section 4.2). For this reason, various decision computation approaches utilize different heuristics to compute approximate solutions with varying degrees of success. The following subsection discusses in technical detail the state-of-the-art approaches and highlights the limitations upon which this Chapter aims to improve.

4.2 Existing Approaches

This subsection compares and contrasts existing object topology computation algorithms using Definition-2 in section 4.1. While most of these works were discussed in Chapter 2, a more technical perspective is adopted in this section so as to better understand the technical significance of the solutions proposed in this chapter.

As discussed in Chapter 2, computations of object topology decisions are performed on abstract representations of the runtime behaviour of an application’s objects. These abstract representations can either be of fine (object-level) or coarse (class-level) granularity, each of which presents a trade-off between the efficiency of the decision computation process and the efficacy of the topologies generated, as discussed further in the following two subsections.

4.2.1 Object-level Granularity

As was introduced in Chapters 2 and 3, in object-level adaptation (Rossi P. and Ryan C., 2005), adaptation decisions are performed by computing and ranking the utility of placing each object to each remote device based on two factors: 1) The degree to which the source node’s load can be mitigated and the load difference with the target device reduced (for each metric type) as expressed in equation 1. 2) The performance improvement that is attainable as a result of migrating a given object to a candidate device, as expressed in Equation 2. These two factors are computed using software and environment metrics proposed by (Gani, 2010; Ryan C. and Rossi P., 2005). The
results of these two factors are aggregated with the highest scoring object to device placement option selected and implemented. The process is repeated until either all mobile-objects are migrated or an object-to-node match that can achieve a minimum threshold score is no longer available.

Since the placement of every object is separately evaluated and performed, object-level adaptation approaches offer flexible object distributions that allow for a more accurate matching of object resource requirements to external resource availability. This gives rise to topologies that are able to satisfy the requirements of the application objects (objective 1 in section 4.1), the client device (objective 2), and the collaborating external candidate devices (objectives 3).

However, object-level approaches omit coupling information from the decision-making process (discussed in more detail in Chapter 2), which discounts objective 4 discussed in section 4.1 (placement of highly coupled objects together) and consequently yields poor efficacy object topologies. Moreover, efforts to improve the efficacy of object-level adaptation by factoring in object coupling information result in computationally infeasible solutions as will be discussed in more detail in section 4.3. In addition the process of computing the utility of each object’s placement on every device is computationally expensive, and the process unscaleable for heavy applications with larger number of application objects.

In summary, while object-level adaptation offers flexibility in determining object distributions, it results in suboptimal topologies because of the omission of object-coupling information from the decision making process. Furthermore, it incurs performance and resource utilization costs as a result of its fine granularity, which in turn limit its scalability for large applications and collaboration scenarios in mobile and pervasive spaces.

### 4.2.2 Class-level Granularity

In class-level granularity approaches, object topology decisions are computed on abstract representations of applications wherein objects are grouped and abstracted based on their class type, and thus the runtime resource utilization and coupling behaviour of all objects of the same class type are represented as an aggregate. Specifically, the runtime of an application is represented as a dynamic weighted undirected class graph, G(V,E) where each vertex v, in the graph G, represents a class and all its runtime objects, whereas an edge e, represents the coupling between all objects of two class types.
Both vertices and edges are weighted, with the weight of a vertex typically represented as a composite sum of various runtime metrics of a class and all its objects. This is shown in equation (4) (Ou et al., 2006) in which the memory, CPU and network utilization costs of a class are aggregated using different importance weights ($\varepsilon_1$, $\varepsilon_2$, $\varepsilon_3$). On the other hand the weight of an edge is commonly represented as the total method invocation count (Gu et al., 2003; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) between the instances of any two classes and represents the coupling intensity between them. However, this does not accurately reflect the true coupling intensity between objects from the context of computing object topologies, and hence a more accurate approach is proposed and used in this chapter, as is later discussed in section 4.4.

$$w(v) = \varepsilon_1 W_{MEM} + \varepsilon_2 W_{CPU} + \varepsilon_3 W_{BW}$$ (4)

Figure 4-1 A class diagram of a simple application consisting of three classes

A class graph evolves and dynamically updates to reflect changes to the application’s execution behaviour based on collected software metrics. Thus, new vertices and edges are constantly created or removed, and the weights of these elements are frequently updated to reflect the current runtime behaviour of an application. Figure 4-1 illustrates the class graph for an application consisting of three classes, whose vertex weights are composited from the memory and processor utilizations of its classes.

Given this dynamic application graph representation, object topologies can be computed by employing a heuristic graph-partitioning algorithm. As discussed in Chapter 2, the problem of graph partitioning is that of separating a given graph $G$, into $k$ disjoint subsets each satisfying a given constraint, while minimizing the number (and weight) of edges which connect these subsets. The problem is known to
be NP Complete (Garey and Johnson, 1979), and while various heuristics have been proposed for different domains, they are mostly designed for resource copious devices and static graphs and are thus not applicable to resource constrained heterogeneous environments. Hence, various works in Adaptive Computation Offloading have offered resource efficient derivatives of these heuristics, of which the current state of the art approach (Ou et al., 2006), as identified in Chapter 2, is discussed in the following paragraph. Note that since a detailed discussion of general graph partitioning techniques is not requisite to the focus of this Chapter, it is deferred to the background section of Chapter 5, where such a discussion is more pertinent.

The current state-of-the-art approach for computing class-level object topologies is that of Shumao et al. (Ou et al., 2006) which proposes a heuristic application graph partitioning approach derived from the original Multilevel graph partitioning algorithm presented by (Karypis and Kumar, 1995). The approach allows adaptation across multiple constrained devices by successively coarsening an application’s class graph which is done by randomly selecting vertices and merging them with their lightest (low vertex weight) but highly coupled (high edge weight) neighbour. The process is repeated until the number of vertices is equivalent to the number of collaborating devices at which point each vertex in the resultant coarse graph is mapped to a device in the collaboration and represents a partition of the original graph. The coarsest resultant graph, which represents the partitioned topology, consists of only as many vertices as there are collaboration nodes, with each vertex or partition mapped to a single node within the collaboration (see Figure 4-2).

Constraints (thresholds) are placed on the size of each partition/subset, which influences the total weight and number of classes that can be grouped within it. These
constraints typically correspond to either resource availability of a candidate remote node or the amount of resource mitigation required by the adapting device. The edge-cut in the partitioned class graph thus represents an estimation of the total network cost (due to remote procedure calls) that would occur in the new object topology and is used as a measure of the efficacy of an adaptation as discussed in section 4.1.

The algorithm ensures that the weight of each subset in the final graph, which is the cumulative resource usage of its constituents, does not exceed the constraints of its corresponding device, thus avoiding overloading the candidate (Objective 3) and achieving the client’s criteria (Objective 2) while meeting the application’s resource requirements (Objective 1). In addition, unlike object-level adaptation, the approach reduces inter-device communication cost by co-locating highly coupled classes, thereby accounting for objective 4 (discussed in section 4.1) of the adaptation process.

Unlike object level approaches, the performance gain offered by migrating each class (as determined by the predicted performance calculation discussed in section 4.2.1) to a remote device is not evaluated separately. While this is largely a result of the graph partitioning strategy employed by existing work, it could easily be addressed by modifying vertex weights within an application class graph to be contextual depending on the partition (device) on which they are placed during the graph-coarsening phase. Thus, the decision to assign a vertex to one partition instead of another could be based on the predicted weight of a vertex on the specific device abstracted by a partition instead of its fixed vertex weight. Hence, the algorithms discussed above and proposed in this Chapter are generic and can be extended to accommodate different adaptation objectives in future, through different vertex and edge weight policies (fixed weight or contextual weight). Thus, for simplicity, load mitigation objectives are discussed and evaluated in this Chapter.

Limitations of Class-level Adaptation Decision Computation:

Despite its clear benefits relative to object-level adaptation, class-level approaches have limitations caused by the smaller number of classes relative to the number of objects. This is because for any application with a set of objects $O$ and a set of loaded classes $C$, the inequality $|O| \geq |C|$ holds, with the only case in which $|O| = |C|$ occurring if all objects are singletons (Gamma et al., 1995). The difference between the number of objects and classes would be more pronounced in computationally heavy applications, which would be expected to create higher number of objects. For example, in a Java based implementation of the Barnes-Hut algorithm (Cahoon and McKinley, 2001), 8 classes are loaded but over 2408 objects
are instantiated. Similarly during a simple run of the NASA WWJ (National Aeronautics and Space Administration, 2004) application (discussed in more detail in section 4.5) only 80 classes are loaded but over 12,000 objects are instantiated. This difference between the number of classes (vertices) and the number of objects they abstract, means that class graphs offer very coarse granularity as compared to objects-level approaches. Consequently, these class graphs have relatively heavy vertices and edges as the large number of objects and inter-object relationships are mapped to fewer class level vertices and edges. This in turn limits the flexibility of adaptation and the efficacy of object topologies generated as a result of the specific limitations listed below:

a) Limited adaptation possibilities: Adaptation in constrained collaboration environments becomes challenging since clients cannot adapt to other constrained nodes which offer resources that are less than the aggregate size of a class (for example, average class memory utilization of 4MB in the case of the Barnes-Hut application mentioned earlier). This is because the migration of even a single class may not be possible on constrained or loaded devices.

b) Reduced Efficacy of Topologies: The large edge weights in the graph potentially result in partitions with large edge-cuts and consequently high inter-node network communication cost. This is because a single edge represents the aggregate invocation cost of all objects of both classes; this means that placing any two classes apart incurs the same cost as placing all their objects apart.

c) High migration cost: Migration of a class involves the migration of all its instances, which could be bandwidth intensive for large classes with many instances.

In summary, while class-level adaptation approaches are more computationally feasible and offer better efficacy than their object-level counterparts, their coarse-granularity reduces the flexibility of object topologies and thus limits the efficacy of adaptation decisions attainable. This in turn incurs additional power, performance and network overheads, which limits the utility of class-level approaches in mobile environments.
4.3 Preliminary Investigations

From the discussions presented in section 4.2, the following two hypotheses are drawn, which are explored in more detail in this subsection so as to cogently approach the solution proposed later in section 4.4.

**Hypothesis 1:** A coupling guided object-level adaptation would provide more efficacious object topologies than class-level approaches

An ostensible solution that emerges from Hypothesis 1 is an object graph based approach to adaptation, which merges the fine-granularity of object-level decision computation with the coupling guided adaptation of graph based approaches. In this approach the runtime application state would be modelled as an object graph consisting of vertices representing individual objects, and edges representing inter-object coupling patterns. Though the concept itself is not original, having been briefly outlined by some works in class-level adaptation (Gu et al., 2003; Ou et al., 2006), no implementation or empirical evaluation of the efficiency or efficacy of such a technique has been provided. Hence, an implementation and evaluation of object graphs was explored by the present author using a corpus of real-world applications under evaluation scenarios discussed in section 4.5.

However, contrary to the expectations of Hypothesis 1, it was noted that the edge-cuts of the generated object topologies were in some cases inferior to those of their class-level counterparts. This is because class graphs conceal the coupling between objects of the same class, which is beneficial for highly self-coupled classes, wherein a concealed self-coupling edge means that regardless of the topology computed highly self-coupled objects do not cause inter-node communication. This is not the case in object-level adaptation in which such edges are exposed and could end up across different partitions (self-coupling is discussed further in the context of the proposed approach in section 4.4)

**Hypothesis 2:** Computational feasibility or efficiency depends on the number of components and their representation i.e. more components and more complex representations (in which coupling information is considered) reduces adaptation efficiency

As discussed in section 4.2.1, object level approaches generally incur higher decision computation cost as a result of the number of element abstractions that had to be considered during decision-making. This performance cost is exacerbated in the case of the object graph approaches (explored for Hypothesis 1 above) in which object coupling information had to be accounted for.
In general, it was observed that the size and structural complexity of object graphs made them computationally expensive to maintain and partition. For instance, the Java based Barnes-Hut algorithm in the J-Olden Benchmark suite (Cahoon and McKinley, 2001), when executed with basic input parameters, produces a class graph comprising 8 vertices and 15 edges with an average degree of 4, compared with an object graph of 2,408 vertices and 3,604 edges with an average degree of 2.99. Consequently, the class graph required 2.6 Kilobytes in memory to maintain and took approximately 27ms to partition, whereas the object graph occupied 746 Kilobytes in memory and took over 30 minutes to partition on a typical smartphone device (Android based HTC Dream with 528 MHz processor). To further aid in understanding the disparity between the two granularities, Figure 4-3 and Figure 4-4 graphically illustrate the class graph and object graph of the Barnes-Hut algorithm respectively, through which the relative complexity and size of an object graph is further emphasized.

Figure 4-3 Barnes-Hut implementation class graph $|V|=8$, $|E|=15$
 CHAPTER 4 Improving Object Topology Computation Efficacy

**Figure 4-4** Barnes-Hut implementation object graph $|V| = 2408$, $|E| = 3604$

**Hypothesis 3:** Given Hypothesis 1, and 2 the best approach for computationally heavy applications would be a hybrid approach between class and object-level granularities, which balances adaptation efficacy with computational efficiency.

Hence, it becomes evident that a better solution to this problem is to offer more adaptation flexibility than class level approaches while reducing the computational limitations of object-level granularities. To this end, the following section proposes and discusses a Hybrid Granularity Graph approach for computing adaptation decisions.

4.4 Hybrid Granularity Graphs

In this section, a novel Hybrid Granularity Graph (HGG) for representing an application’s runtime is proposed. In this hybrid graph approach, a vertex does not necessarily map to one runtime component (e.g. Object or Class) but rather to a configurable subset of objects of a given class, while an edge attached to any two such vertices would represent the total coupling between all objects in the two subsets. Hence, HGG offers finer granularity, and consequently more flexibility, than a class graph and yet remains smaller, and consequently more computationally...
feasible, than the corresponding object graph. As a result HGG would contain lighter vertex and edge weights as compared to a class-level graphs thus allowing for greater flexibility and consequently more efficacious adaptation, as will later be discussed in the evaluation in section 4.4.

Since such a graph does not directly map to a component in the running application, it must be derived from the graphs of one of the two levels of granularity. Hence, two possibilities for dynamically generating such a graph were identified: 1) Compacting an object-level graph so as to create a smaller, coarser grained graph or 2) Decomposing a class-level graph so as to create a relatively finer-grained graph. Each of these approaches is discussed in the following subsections.

![Figure 4-5 Granularity Decomposition Diagram](image)

### 4.4.1 Object Graph Compaction

Constructing a Hybrid Granularity Graph from an object graph involves successively merging its vertices so as to reduce the size of the object graph (thus increasing its granularity) until a computationally feasible graph size emerges. This successive merging of vertices is based on a number of criteria, which are listed and discussed below:

**Criterion 1: Merge objects of the same class type:** In a Hybrid Granularity Graph, each vertex would homogenously contain objects of the same class type. Hence, the object compaction process must ensure that objects of the same class types are identified and subsets of these objects merged into a single vertex that abstracts their cumulative resource usage. However, the merge must ensure that the total size of the objects merged does not exceed a specified threshold so as to ensure that heavy vertices do not emerge in the final hybrid graph (discussed in criterion 3 below).

**Criterion 2: Merge Highly Coupled Objects:** As discussed in section 4.3, the primary reason for which object graphs resulted in poor efficacy adaptation decisions relative to their class-level counterparts was the exposure of self-coupling edges in object graphs which were otherwise concealed in the class-level graphs. Hence, the object-graph compaction approach must ensure that the same limitation doesn’t emerge in the Hybrid Granularity Graph. This can be achieved by prioritizing the merge of
highly coupled objects (of the same class type) during the successive merging phase discussed in criterion 1 above. Such an approach would hide heavy self-coupling edges in the resulting HGG, while ensuring the co-locality of such highly coupled objects.

**Criterion 3: Bounded Vertex Weights:** Given that one of the primary objectives of an HGG, is the creation of an intermediate level graph that is neither too fine nor too coarse in granularity, the vertex merging processes discussed in objectives 1 and 2 above, must ensure that the weight of each aggregate vertex (whose weight is the cumulative weight of its constituents) does not exceed a specified threshold $T_v$. This is so that vertices in the resulting graph do not consist of large vertex weights, which would increase its coarseness and reduce its flexibility, as is the case with class-level graphs.

**Criterion 4: Bounded Edge Weights:** Similar to criterion 3, it is noted that the merging of vertices also results in the unification of their edges, which could potentially lead to heavy edge weights and potentially inflexible adaptation, as is the case in class graphs. Hence, again similar to objective 3 above, the merging of vertices during the compaction process proceeds only if the sum of the resulting cumulative edges does not exceed a specified threshold $T_e$. This is done so as to ensure the resultant graph does not possess heavy edges that limit the flexibility of the resulting HGG, as is the case in class graphs.

The above criteria could further be extended to include other factors where for instance only objects with similar resource requirements could be merged together. This could be beneficial when objects of the same class type utilize different resources to varying degrees (some might be memory intensive, others processor intensive etc.). Hence merging objects with similar resource type requirements would allow a more accurate matching of their requirements to available resources during adaptation.

However, prima facie, it is noted that an object graph compaction approach for generating an HGG would be computationally expensive as it involves the maintenance, analysis and processing of a potentially large graph (discussed in section 4.3) before achieving the desired level of granularity. The process is thus not ideal in mobile and pervasive environments where the costs of deriving an HGG could potentially outweigh its gains. Hence, a more computationally feasible alternative is presented in the following subsection.
4.4.2 Class Graph Decomposition

In the class graph decomposition approach, selected classes are fissured into two or more vertices based on a set of criteria. The edges of the original vertex are also fissured and each resultant fissured vertex shares the same coupling patterns as the original with a proportional fraction of the original edge weights. In order to ensure that the resultant HGG offers more efficacious partitions (i.e. lower edge-cuts), while preserving the computational feasibility of a class graph, the decomposition needs to be controlled based on specific criteria for how vertices are selected for fissuring and the manner in which they are fissured.

In general, the graph decomposition should:
1) reduce the average weight of a vertex in the graph by fissuring as many large vertices as possible; 2) reduce the total additional edge weight that results from the graph decomposition, wherein additional edges arise from the self-coupling of fissured classes; and 3) ensure that the resultant graph size is not computationally expensive to process as per the device’s capabilities. Hence, the specific set of criteria for fissuring vertices in order to decompose a class-level graph into a HGG are stated below:

Given that one of the primary motivations when deriving an HGG from a coarse-grained class graph is to achieve finer granularity, the weight of vertices (which in Figure 4-6 above is shown as the memory and processor utilization of a class) is an important factor that must be optimized. Hence, assuming a vertex contains more than one object, it is considered for fissuring only if its weight exceeds a specified threshold $T_w$. This threshold can be established empirically through observation of an

Figure 4-6 Example Class Graph showing criteria 1-3 from the perspective of a heavy class X

**Criterion 1: Large Vertex Weight**

Given that one of the primary motivations when deriving an HGG from a coarse-grained class graph is to achieve finer granularity, the weight of vertices (which in Figure 4-6 above is shown as the memory and processor utilization of a class) is an important factor that must be optimized. Hence, assuming a vertex contains more than one object, it is considered for fissuring only if its weight exceeds a specified threshold $T_w$. This threshold can be established empirically through observation of an
application’s resource usage (during runtime metrics collection) and adaptively calibrated as will later be discussed in section 4.4.4.

**Criterion 2: Low Self-Coupling Edge Weight**

Each vertex in a class graph has an implicit loop edge, which represents the coupling amongst objects of the same class as shown in Figure 4-6. While this does not impact the topology generated by a class graph, it determines the extent to which large self-coupling edges, which were otherwise concealed, are exposed in the HGG. This is because fissuring a vertex results in this self-coupling edge being exposed as one or more connecting edges for the newly fissured vertex (this is later visualized in Figure 4-8 and Figure 4-9 in page 89); thus directly influencing partitioning decisions, recalling how the exposure of high self-coupling edges in object graphs resulted in lower efficacy in some cases as was discussed in section 4.3.

Therefore, in order to avoid suboptimal decisions, a vertex is fissured only if its self-coupling edge has a relatively smaller weight than the coupling edge with its neighbours (note that this is not the case for the example diagram shown in Figure 4-6). In such a case, fissuring a vertex would introduce a new light edge to the graph, which would allow for the possibility of more efficacious partitions while also reducing the total increase in edge weights resulting from the decomposition. Specifically, the self-coupling weight of a vertex has to be smaller than all other edge weights of the vertex by at least $T_e$, in order to be considered for fissuring.

**Criterion 3: Low Vertex Degree**

Each time a vertex, $v$, is fissured, the number of edges $|E_0|$ in the original Graph $|G_0|$ increases. For example, by as much as $\deg(v) + 1$ in the case where the node is fissured into two parts, and increasing with the number of parts into which the vertex is fissured (this is discussed further in section 4.4.3). Hence, in order to achieve as many fine granularity vertices as possible before reaching the graph size limit due to edge growth (see criterion 4 below), only vertices with degree sizes below a specified threshold $T_d$ are considered for fissuring.

**Criterion 4: Bounded Graph Size**

The graph decomposition should ensure that the HGG remains computationally feasible and thus both a vertex and edge size limit is imposed on the evolving class graph and are specified as the threshold values $T_{|V|}$ and $T_{|E|}$. 
Vertex-level criteria, 1-3, define the eligibility of each vertex based on its non-functional behaviour (i.e. resource usage, performance etc.) as shown in expression 5 below, whereas Criterion 4 is a graph-level criteria and defines properties and restrictions of the resultant HGG, as shown in expression 6.

\[
V_{fissurable} = \{v: v \in V_0, [w(v) \geq T_w] \land [\text{deg}(v) \leq T_d] \land [\forall u \in V \setminus v \text{ and } e_{u,v} \in E_0; w(e_{u,v}) + T_c \leq w(e_{u,v})] \}
\]  

Expression 5 above shows that the set of all fissurable vertices within the graph should satisfy the vertex weight constraint (\(T_w\), criterion 1), the self-coupling constraint (\(T_c\) criterion 2) and the degree constraint (\(T_d\), criterion 3). While the above four criteria are used to determine fissurability of vertices based on their non-functional properties, they can be extended to include other factors. For instance, a criterion could be included to determine fissuring of classes with constituents of conflicting mobility constraints such as a Character class in a game application, which might need to migrate its objects to different nodes for improved functionality (for example, placing each Character instance in a multiplayer game on the device of the player who is controlling it). Note this would not be possible in class level adaptation as all Character instances could only migrate to the same device. Similarly other non-functional criteria could be considered, whereby for instance fissuring could be performed on classes with high resource utilization and coupling pattern divergence among their constituents, so as to create HGGs with homogenous vertex constitutions which would be expected to improve computed object topologies in terms of the accurate matching of resource requirements to resource availability and the reduction of edge-cuts.

The vertices from within this set that are fissured are subject to the graph level constraint (criterion 4) as shown below.

\[
\text{Decompose}(G_0) \rightarrow G_1(V_1, E_1) \text{ where } \forall v \in V_1, \hfill \hfill
w(v) \leq T_w + \delta_{|V|} \lor |V_1| \geq |V| - \delta |V| \lor |E_1| \geq |E| - \delta |E| \hfill \hfill
\]  

Expression 6 shows that the HGG emerging from the decomposition of the class graph \(G_0\) can be described using its expected granularity (Criterion 1) and resultant graph size (Criterion 4). Specifically, the resultant HGG would have either all its vertex weights conforming to the granularity specified by \(T_w\) or has achieved its

---

3 Objects of the same class having different resource requirements, for instance certain objects being more memory intensive while others are processor intensive etc.
maximum allowed size defined by $T_{|V|}$ or $T_{|E|}$, where $\delta_{|V|}$, $\delta_{|E|}$, $\delta_w$ define the allowed flexibility from the specified threshold values $T_{|V|}$, $T_{|E|}$ and $T_w$.

### 4.4.3 When to Decompose a Class Graph

It is identified that there are three possible stages during which the class graph decomposition could be performed:

1) **During Metrics Collection**

As discussed earlier in section 4.2.2, the class graph maintained for an application is dynamic and hence the graph elements and their weights are constantly updated to reflect the behaviour of the components they abstract. This process of updating the runtime class graph is performed in accordance with the metrics collection phase, whereby a device (through its underlying adaptation middleware) periodically (or in an event triggered manner) measures the resource usage of components and the environment as discussed in Chapters 2 and 3.

Hence, one approach for creating a Hybrid Granularity Graph is to gradually decompose a class graph during the metrics collection phase. In this approach, an HGG generation algorithm would evaluate the fissurability of a vertex as soon as its weight is adjusted by a metrics collection system (discussed in Chapter 2), and thus the class graph gradually evolves into an HGG as the application’s execution progresses.

The primary advantage of this approach is that it merges the graph decomposition phase with the graph update phase, which occurs when new metrics are collected. This means that a separate traversal or processing of the class graph for the sole purpose of decomposition is not required, thus reducing the overheads associated with the generation of an HGG.

However, a major limitation of this approach is that the graph size upper bound for an HGG (criterion 4 discussed in 4.4.2) could be achieved prematurely, after which case further changes in an application, which might involve the emergence of heavier vertices (relative to already fissured vertices), would remain un-fissured. Thus, the resulting graph, though not as coarse as a class graph, would contain a skewed granularity consisting of potentially heavy vertices alongside fine granularity vertices. This presence of potentially large numbers of coarse granularity vertices is not ideal for reasons discussed in earlier subsections, and hence alternative approaches are presented below. In addition, section 4.4.5 discusses an elastic
granularity approach, in which a graph could re-coarsen its granularity to better address this limitation.

2) **Pre-Adaptation Time**

In this approach, the graph is decomposed immediately prior to performing an adaptation decision computation. The process involves traversing the runtime graph so as to fissure vertices which meet the criteria discussed in section 4.4.2, as shown in the algorithm in Figure 4-7 on page 88. The advantage of the approach is that it reduces the probability of prematurely achieving the graph size threshold by processing the overall graph and deferring the process of decomposition until the adaptation phase, where the granularity of an adaptation is of importance, unlike the approach discussed in 1) above.

However, unlike the previous strategy, this approach requires a separate phase for traversing and processing the class graph and thus is more computationally costly than the approach discussed in 1) above and 3) below.

3) **During Adaptation Decision Computation**

Another approach to decomposing an application class graph is during adaptation decision computation, where vertices are fissured while in the process of computing an application graph partitioning. Since the approach would largely be dependent on the graph-partitioning algorithm utilized, the derived multilevel graph partitioning approach proposed by Shumao et al. (Ou et al., 2006), which was briefly discussed in section 4.2.2, is used below as an example of how this technique can be implemented.

In such a case, an HGG would be created during adaptation decision computation by merging the graph decomposition phase with the coarsening phase. Specifically, prior to merging a neighbour to a randomly selected vertex (vertex matching phase), its fissurability is evaluated, after which time it is fissured if determined to be eligible according to the criteria discussed in section 4.4.2. Once a fissuring occurs, the utility of merging each fissured neighbour vertex is re-evaluated by considering its new weight and coupling intensity to the randomly selected vertex. This means that adaptation decisions are performed on the finer Hybrid Granularity Graph as opposed to the class graph on which the adaptation decision commenced.

Since the approach combines the graph decomposition process with the graph partitioning phase, it reduces the need to separately traverse and process the graph, thus resulting in less computational overhead as compared with 2) above. In addition, the approach defers deriving the hybrid graph until it is required for adaptation, thus reducing the runtime overheads associated with an HGG (discussed in section 4.5),
unlike the first approach (during metrics collection time). Furthermore, it allows for the creation of a hybrid granularity graph based on attributes of relevance to the specific partitioning decision (in-terms of partitioning constraints) being performed. For instance, during the final stages of the coarsening phase, vertex-fissuring decisions could be performed if they serve to minimize the resulting edge-cut between partitions and achieve the constraints of the collaborating devices.

However, so as to broaden the applicability of this work, only the second approach is discussed in this Chapter. This is because unlike the other approaches, 1) and 3) above, it is independent of the selected graph update policy and underlying graph partitioning technique, thus allowing for independent analysis of the HGGs efficacy.

The HGG generation algorithm provided in Figure 4-7 provides a pseudocode implementation of the criteria discussed earlier. The algorithm traverses the class graph (lines 5-14) fissuring vertices, which satisfy criteria 1-3 (line 18,19) while ensuring the graph does not exceed its bounded size (line 20-22) as specified by the defined thresholds (lines 1-3).

Figure 4-9 shows a fissuring of the vertex for class1 in Figure 4-8, into two vertices class1’ and class1”. The figure shows how the graph evolves in this scenario with the introduction of a new edge, which represents the coupling between the two
vertices. The hybrid graph results in lighter vertices and edges, which are expected to improve flexibility during decision making, as evaluated in the following section.

Figure 4-8 Class Graph consisting of three vertices with a single heavy vertex

Figure 4-9 Fissuring of class1 (from 0) into class1’ and class1’’
Figure 4-10 NASA WWJ Class Graph

Figure 4-11 NASA WWJ Hybrid Granularity Graph. Fissured vertices are shaded black.

To further assist in visualizing HGG, Figure 4-10 and Figure 4-11 illustrate the class graph for the NASA WWJ application (National Aeronautics and Space Administration, 2004) and its associated derived Hybrid Granularity Graph, in which dark vertices represent fissured classes. It can be seen that the HGG graph (Figure 4-11) contains more vertices and yields less heavy vertex weights (as reflected by the size of the vertices) in comparison to the coarse class graph visualized in Figure 4-10.

4.4.4 Prioritized Vertex Fissuring

While expression 5 in section 4.4.2 showed a rule-based scheme to determine the fissurability of an individual vertex, a more sophisticated approach could also be adopted so as to maximize the number of vertices fissured before reaching the graph level bounds (criterion 4 in section 4.4.2). In this approach, a score is first computed for each vertex based on the degree to which it meets each of the criteria 1-3 in section 4.4.2, as shown in equation 7 below. The equation computes a value between [-1,1] for each criterion \( c_i \) (over the set of all values \( C_i \) for the criterion) wherein positive values indicate that a vertex achieves a specified criterion to a higher degree and negative values indicate conditions in which a vertex does not meet the specified threshold values for a specific criterion. Once each criterion score is calculated a weighted sum of the scores is computed to determine the fissurability of a given
vertex as shown in Equation 8. The weight of each criterion indicates its importance in computing the final score, wherein for instance criterion 3 could be weighted higher in order to increase the number of fissures that can happen before reaching the graph bounds (criterion 4). Finally, vertices are ranked based on their overall fissurability score so as to prioritize and process more desirable elements (as per the criteria weights employed) before reaching the graph size bounds enforced by criterion 4.

\[
s_i = \frac{c_i - (T_i + \delta_i)}{\max(C_i)}
\]

(7)

\[
S = \frac{\sum_{i=1}^{\mid C \mid} w_i s_i}{\mid C \mid}
\]

(8)

4.4.5 Elastic Hybrid Granularity Graph

In addition to the vertex fissuring criteria and graph decomposition strategies discussed above, one approach that could be used to offer dynamic and more fine-grained control over adaptation granularity is an Elastic Hybrid Granularity Graph approach. In this approach an application graph is not only modified towards the direction of an HGG (as is the case in Class Graph Decomposition and Object Graph Compaction), but is performed as a dynamic two-way process whereby its granularity is coarsened or de-coarsened as required. For instance, a class graph could be decomposed so as to leverage the benefits of finer granularity during an adaptation decision (as discussed in the above subsections), and then later composed back into a class graph so as to minimize the overheads of maintaining an HGG (the overheads of HGGs relative to class graphs are discussed in section 4.5).

Hence, this approach allows for the optimization of various factors such as the resource cost of the adaptation process and the efficacy of its decisions. In addition, the approach could be used to facilitate the creation of different ranges of Hybrid Graphs with varying degrees of granularity, by adaptively calibrating threshold values for the criteria discussed in section 4.4. Such an approach would offer dynamic and more fine-grained control over the granularity of an application’s graph, which would be expected to create higher efficacy adaptation results while minimizing the overheads of adaptation.

However, while these facets present interesting avenues for further exploration, a thorough exposition and evaluation is left for future work.
4.5 Evaluation

This section discusses the experimental evaluations performed, and the results obtained in comparing the efficacy and computational efficiency of the proposed hybrid granularity graph with object and class level approaches.

To this end, subsection 4.5.1 first provides an overview of the tools and methodologies employed in implementing the algorithms discussed in this chapter. This is followed in subsection 4.5.2 with an outline of the experimental materials selected in terms of the test applications, experimental devices and the underlying network infrastructure. These experimental settings are used in subsection 4.5.3 to discuss the adaptation scenario simulated during the evaluation, which is expressed in terms of resource fluctuation patterns and device constraint configurations.

Evaluation results are presented in proceeding subsections, where subsection 4.5.4 provides the first such insight through descriptive statistics of the properties of the generated application graphs (Class, Object and HGG) followed by subsection 4.5.5 which discusses adaptation decision computation efficiency aspects. Finally, subsections 4.5.6 evaluates the adaptation efficacy of the proposed HGG approach and discusses its implications relative to the results presented in section 4.5.5.

4.5.1 HGG construction

A Java based toolkit that profiles an application through bytecode injection and the Java Virtual Machine Tool Interface (JVMTI) (Oracle Corporation, 2004) was implemented. The toolkit recorded performance cost and memory utilization as well as runtime coupling patterns, which were used to construct runtime graphs at both class and object-level granularities. The approach proposed by Shumao et al. (Ou et al., 2006) was the basis for computing the composite weight of the performance and memory costs of an application’s components. However, to better estimate the cost of placing two components apart, the serialized size of parameters recorded during method invocation was used to estimate coupling cost, as opposed to the invocation count used in (Ou et al., 2006). In addition, the proposed Hybrid Granularity Graph generation algorithm and the multilevel graph partitioning heuristic proposed in (Ou et al., 2006) were implemented in the toolkit.

4.5.2 Software Corpus and Experimental Device

To increase the practical utility of the results, real world Java applications were used in preference to synthetic test cases. Specifically four open source applications
(see Table 4-4) and the application demo of the NASA World Wind SDK (version 0.6.5) were selected for evaluation.

To provide a typical example of a constrained device adapting in a mobile environment three devices were chosen 1) an Android based HTC Dream (G1) smartphone with a 528 MHz processor and 192MB RAM served as the adapting client device 2) an HP iPAQ PDA running windows mobile 6.1 with 664MHz processor, and 128MB RAM, 3) a Laptop running windows 7 ultimate on an Intel core 2 duo 2.5GHz processor with 4GB of RAM. All devices were connected over an IEEE 802.11g wireless router.

4.5.3 Adaptation Scenario

In order to simulate adaptation under varying resource availability constraints, a gradually increasing constraint of the Android device was simulated, which forced it to incrementally offload 5% of the application to the remote adaptation targets (HP iPAQ and laptop) until it had either offloaded 95% of the entire application, or had migrated all mobile components (i.e. non-UI and non-entry point components which do not access device specific resources such as databases or sensors) of the application.

Each time the device adapted, the remote targets cumulatively offered the exact level of resources required. Specifically if during its fourth incremental adaptation the device sought to offload 20% of an application, the target devices would offer only as much as \((20\% + \pi)/2\) computing resources (where \(\pi\) is a configurable degree of flexibility) i.e. each peer offered approximately 50% of the required computing resources. This is to simulate the likely case where devices have limited spare resources because they are either simultaneously serving other clients, or predict such usage in the future. Such a scenario also mirrors the case when adapting to a cloud-computing instance in which the client has to pay for additional computing cycles and is thus conservative with offloading.

4.5.4 Generated Hybrid Granularity Graph

Table 4-4 lists the five applications that were used in this study as well as the decomposition constraints (thresholds \(T_w, T_d\) etc. in section 4.4.2) for each application. Note that these were kept constant for the first four applications, whereas a different set of parameters were used for the NASA WWJ application (see Table 4-4 for details), which were more applicable to its computational requirements in
order to limit the resultant graph size. In contrast, the upper bound threshold for self-coupling criteria \((T_c\) in section 4.4.2) of a class considered for fissuring was kept constant for all applications. All constraints were established empirically based on observation of the application graphs. Determining how these constraints could be automatically generated and adaptively calibrated based on application behaviour and the resource availability within the collaboration is left to future work.

Table 4-4 also lists the vertex and edge counts for each application, for class, HGG and object-level granularities. Though the graphs exclude Java API classes, which are assumed stationary (i.e. non-mobile objects), their resource utilization and performance costs are implicitly represented by the application components that utilize them. It can be seen that the HGG has as much as double the number of vertices of the class graph (in the case of Voronoi and E3D applications), but is still more comparable in size to a class graph than an object graph (which is as much as 300 fold larger in the case of the Barnes Hut Algorithm).
CHAPTER 4 Improving Object Topology Computation Efficacy

4.5.5 Results – Object Topology Computation Process

This subsection evaluates the efficiency of computing an adaptation decision as measured through the overheads of maintaining, partitioning and, in the case of HGG, decomposing an application runtime graph. The evaluations compare the memory (section 4.5.5.1), performance (section 4.5.5.2) and power (section 4.5.5.3) costs of the different levels of granularity against that of the proposed HGG approach.

4.5.5.1 Memory Utilization

The memory utilization of an adaptation approach is largely dependent on the number of abstraction elements maintained for an application. Thus, the memory cost of a graph-based adaptation approach of any granularity type (Class, Hybrid and Object) is given as the total memory utilization of vertices and edges within the graph as shown in Equation (9). On the other hand, the memory utilization of the approach proposed by Rossi and Ryan (Rossi P. and Ryan C., 2005) (discussed in section 4.2), which is presented here for comparative purposes, is expressed by the memory cost of the abstraction elements maintained for each object and hence excludes the overheads of maintaining object coupling information, as shown in Equation (10).

\[
mu(G(V, E)) = \sum_{i=1}^{|V|} mu(v_i) + \sum_{i=1}^{|E|} mu(e_i) \text{ where } v \in V \text{ and } e \in E \tag{9}
\]

\[
mu(G(V, E)) = \sum_{i=1}^{|O|} mu(o_i) \text{ where } o \in O \tag{10}
\]

Figure 4-12 shows the memory utilization of the application graphs maintained. It is observed that the class graph approach consumed the least amount of memory for all the experimental applications in comparison with the HGG, Object Graph, and the Object level approaches (Rossi and Ryan (Rossi P. and Ryan C., 2005)) as would be expected from the graph statistics shown in Table 4-4 of section 4.5.4. However, it is also observed that HGG incurred relatively less overheads in comparison to object graph and object level approaches, and was consequently more comparable to class graph approaches than either of the fine grained approaches.

Specifically, HGG incurred the least overhead in the case of the NASA-WWJ application, with memory cost of 1.2 times the size of the class graph, in comparison to 134 times in the case of the object graph based adaptation and 100 times for the object level adaptation for the same application.

Similarly in the case of the Barnes-Hut application, HGG incurred as much as 2.5 times more memory utilization of the class graph as compared to 284 times in the
case of object graph and 216 times in the case of object level adaptation approaches. On average (for all applications) HGG incurred 2.1 times the memory cost of the corresponding class graph in contrast to an average of 178 times for the object graph and 131 times for the object level based approaches.

Hence in summary, it is noted that HGG incurred modest memory overheads in comparison to class based approaches due to its relatively larger graph size, but
remained orders of magnitude more memory efficient in relation to finer granularity object based approaches.

4.5.5.2 Performance Evaluation

The performance of computing an object topology is a result of the process of computing a partitioning of the application graph. Hence, as discussed in section 4.2, the performance cost of partitioning an application graph is largely dependent on its size.

As would be expected from the larger size of HGG relative to class graph based approaches, it incurs modest performance overheads as shown in Figure 4-13 below. However, HGG is still more comparable to that of a class graph than the significantly more costly object graph. Specifically, on the Android mobile device, both the HGG and class-level graphs took under 60 Milliseconds to partition for all 4 open source applications, in contrast to 3.85 minutes for the least expensive object graph. In the case of NASA WWJ, the class and HGG results are again comparable at 1.5s and 3.2s respectively compared with 1 hour for the object graph. It is noted that the performance costs recorded for HGG include the graph decomposition time which could be reduced by implementing the alternative approaches for decomposing a class graph discussed in section 4.4.3.

![Figure 4-13 Object Topology Computation Performance of Class, Object and Hybrid Granularity Graphs. HGG performance includes decomposition time.](image)

4.5.5.3 Power Utilization
Similar to the performance costs discussed above, the power consumption of an adaptation strategy is a factor of both the decision computation process and the efficacy of the adaptation decisions. While this subsection evaluates the former aspect of adaptation, the latter is discussed in section 4.5.6.3. An evaluation measuring the power consumption of the application graph partitioning phase using Power Tutor (Zhang et al., 2010a) showed that the class graph approach offered less power utilization over HGG for all evaluated applications as shown in Figure 4-14 below. This is because the power cost of the former is a factor of the size of the application graph processed during partitioning. Specifically, the class graph reduced power consumption by between 6% in the case of the Health Simulator application and 53% in the case of the NASA-WWJ.

![Figure 4-14 Object Topology Computation Power Utilization result](image)

**4.5.6 Results – Object Topology Efficacy**

As outlined in Definition-2 in section 4.1, the efficacy of an adaptation decision is determined by the degree to which the resource cost within the collaboration (in terms of inter-device communication) is reduced while achieving the adaptation objectives specified in section 4.5.3. In application graph based adaptation (Class and HGG approaches), this factor can directly be measured through the edge-cut of the graph partitioning decisions performed. Hence, the following subsections discuss the efficacy of the two approaches in terms of the inter-device communication cost which is quantified in terms of network utilization (section 4.5.6.1), the associated remote invocation latency (section 4.5.6.2) and power consumption (section 4.5.6.3). In all cases the adaptation decision performed by both approaches were *valid* (Definition-1 in section 4.1) and met the adaptation objectives stated in section 4.5.3. It is noted
that different adaptation objectives (such as performance improvement of the adapting application etc.) could easily be considered by modifying the partitioning constraints used for adaptation.

4.5.6.1 Adaptation Efficacy - Network Utilization

The adaptation efficacies of class and hybrid granularity graphs are compared in Figure 4-15 in terms of the generated average edge-cuts. These edge-cut values demonstrate the network cost in terms of MBytes of serialized method parameters communicated amongst devices as a result of the placement decisions generated by each approach. Lower edge-cuts not only mean reduced network overheads but also reduced remote method invocation latency, as well as reduced battery usage on the adapting device as a result of fewer remote calls, as will be demonstrated in subsequent subsections.
Figure 4-15 Edge-Cut results for evaluated applications

Figure 4-15 a)-e) show that the HGG approach provides better adaptation efficacy through reduced edge-cuts for all evaluated applications; with edge-cut reductions ranging from 17% for the Barnes Hut Algorithm, to as much as 99% for the E3d modelling application. Similarly, an edge-cut reduction of 52% for the NASA WWJ application, 43% for the Hospital Simulator and 25% for the Voronoi application were recorded. This reduction in edge-cut is attributed to the fact that HGG contains lighter edge weights in comparison to the class graph as discussed in section 4.4.

4.5.6.2 Adaptation Efficacy - Remote Method Invocation Latency

The reduced size of serialized parameters communicated during component invocation in the hybrid granularity approach is shown to reduce overall remote method invocation latency. Specifically, as shown in Figure 4-16, the approach resulted in latency reduction of 18% in the case of the Barnes-Hut algorithm, 30% for the Voronoi application, 42% for the Hospital Simulator, 63% for the NASA-WWJ application, and 99% in the case of the E3D application, as compared to the class level approaches. In addition, the figure shows that the initial overheads incurred by HGG during application partitioning time (section 4.5.5.2) are potentially outweighed by the reduction in latency of application execution offered.
Adaptation Efficacy - Power Consumption

The power consumption as a factor of the efficacy of adaptation decisions generated, was quantified by measuring the power cost of remote method invocations (as a result of the edge-cut). It was observed that HGG consumed considerably less power for all experimental applications in comparison to the corresponding class graph adaptation. Specifically, HGG yielded power usage reduction of between 18% in the case of the Barnes-Hut application and 99% in the case of the E3D application, with an average power cost reduction of 50% compared to class level approaches (shown in Figure 4-17 below). The figures show that the initial overheads incurred by HGG during decision computation and graph decomposition (section 4.5.5.2) are outweighed by the power cost reduction offered as a result of the efficacy of the computed object topology. Specifically, a total power utilization reduction of between 16% in the case of Barnes-Hut algorithm and 99% in the case of E3D were recorded relative to class graph based approach.

![Graph showing power consumption comparison between HGG and Class Graph for different applications.](image.png)

**Figure 4-16 Remote Method Invocation Latency as a result of edge-cut**
4.6 Further Improvements to HGG efficiency

The above evaluations identified the efficacy benefits of HGG relative to other granularity approaches, and discussed how the decision computation overheads of HGG relative to class graphs can be outweighed by its efficacy gains. However, it is noted that additional factors could also be considered to further reduce the efficiency overheads associated with an HGG.

Firstly, graph decomposition could be performed during adaptation time as opposed to pre-adaptation time (which was used during the evaluations) so as to reduce the overheads associated with graph decomposition as discussed in section 4.4.3. Secondly, the size of the resulting HGG could be optimized for efficiency by employing graph elasticity and adaptive criteria (calibration) techniques discussed in section 4.4.4. Additionally, it is expected that these techniques would allow greater control over the resource cost of an HGG and thus further improve its efficacy and efficiency.
4.7 Summary

The overall utility of Adaptive Computation Offloading approaches in resource constrained heterogeneous environments, is largely dependent on the efficacy of adaptation decisions computed as identified in research question B.2) in section 1.3.2. It was shown that the granularity at which adaptation decisions are performed, largely influences both the efficacy and efficiency of adaptation decisions, with existing class and object level granularities resulting in either inefficacious or inefficient adaptation alternatives. Hence, this chapter focused on improving this aspect of adaptation by specifically focusing on the granularity at which adaptation decisions are performed.

To this end, a novel Hybrid Granularity Graph (HGG) was proposed, which combined the efficacy benefits of fine-granularity (object level) with the computational feasibility of coarse-granularity (class level) adaptation. In addition, a novel approach for deriving this level of granularity through the dynamic decomposition of runtime class graphs was outlined.

An evaluation using a small corpus of real-world applications in a heterogeneous collaboration setting, showed that the HGG approach offered improved adaptation efficacy in terms of edge-cut reduction of between 17% and 99% as compared to class level granularity; thus addressing research question B.2) in section 1.3.2. In addition, these improvements in adaptation efficacy translated into reductions in power and performance costs in comparison to class-level approaches, when comparing the effect of the edge-cut (efficacy) improvements in isolation.
Chapter 5  Improving Efficiency and Scalability of Object Topology Computation

As discussed in Chapter 1, the practicality of using adaptive offloading to enable the migration of heavy applications to mobile environments is predicated on the efficiency and scalability of adaptive decision computation algorithms. However, existing approaches to adaptive offloading incur overheads from storing, updating and partitioning complete application graphs on each device, which limits both their efficiency and scalability in resource constrained environments. Hence, this Chapter focuses on addressing these limitations so as to improve the overall utility of adaptation.

To this end, in this Chapter, a novel distributed application graph representation and an associated graph partitioning heuristic are proposed which reduce the overheads of maintaining application runtime graphs and improve adaptation decision computation performance. These improvements were quantified in a laboratory evaluation involving three heavy open-source applications adapting on a constrained mobile device, a desktop class machine and a cloud-computing instance. Specifically, performance gains of between 19.47% and 93%; collaboration-wide memory reduction of between 37% and 50%; network consumption reduction of 100%; and reduced power consumption of between 63% and 93% were recorded. Furthermore, adaptation efficacy (discussed in Chapter 4) was improved by 12% and 34% for two of the three evaluated applications, and remained the same for the other.

The Chapter is structured into five subsections as follows: Section 5.1 first presents a rationale for optimizing the efficiency and scalability of adaptive offloading, by quantifying the overheads associated in an example adaptation scenario. A more technical discussion of the causes of these overheads is presented in section 5.2 through an investigation of existing work, which identifies underlying limitations. These limitations are then addressed through a novel solution, which is proposed in Section 5.3 and evaluated in section 5.4. Finally, section 5.5 provides a summary of the Chapter and discusses some of the implications of the proposed solution from the perspective of the goals of this thesis.
5.1 Motivation

In Chapters 2 and 4 some of the efficiency limitations of current state-of-the-art adaptation approaches were outlined. This section quantifies those limitations and other incumbent adaptation overheads without specifically delving into the processes involved (which is discussed in section 5.2), so as to provide a rationale for optimizing the efficiency and scalability of adaptation decision computation.

In Chapter 4, it was shown that the granularity at which adaptation decisions are computed (object, class or HGG) significantly influenced both the efficacy and efficiency of the application graph partitioning process. Specifically, while finer granularity approaches offered better efficacy they usually resulted in greater computational overheads, primarily as a result of the number and complexity of the abstraction elements involved. This increased computational cost was also reflected in the evaluation of the proposed HGG approach; which although showed greater efficacy, incurred more computational costs than the simpler class graph based representations. With a cubic runtime complexity (section 5.2.2) of the most efficient adaptation approach, the increase in application graph size often resulted in considerable reduction in efficiency as was shown in section 4.5. Hence, it was shown that efficiency (of object topology decision computation) and efficacy (optimality of a computed component topology) are trade-offs and thus in order to leverage the efficacy gains of finer granularity approaches (HGG), more scalable decision computation strategies are requisite.

Similarly, it can be observed that adaptation approaches are less scalable to more computationally heavy applications, which are expected to have larger number of elements with more complex interaction patterns (Abebe and Ryan, 2011b; Ou et al., 2006). This observation can be quantified by considering three separate applications $A_1$, $A_2$ and $A_3$ of different sizes, in a hypothetical adaptation scenario involving three heterogeneous devices, ($D_1$ Android based HTC G1 Smartphone and $D_2$ and $D_3$ Desktop class machines) adapting over an IEEE 802.11g wireless link. The applications represent different sizes and coupling complexities as is shown in Table 5-1, with the smallest application being $A_1$ and the largest application with the most complex coupling behaviour being $A_3$.

In a scenario in which near equal portions of the applications were distributed across each device, and device $D_1$ ran out of resources and computed an adaptation decision (using approach discussed in section 5.2.2), it was observed that the decision computation times increased considerably relative to the increase in application size.
Specifically, it took 33 times longer to adapt $A_2$ than $A_1$ despite an only 6-fold increase in the number of classes of $A_2$ relative to $A_1$. Similarly, the adaptation decision computation time for $A_3$ was 140 times longer despite an only 18-fold increase in class size relative to $A_1$. In addition, the cost of increased application size was also reflected in the resource utilization of the applications during adaptation. Specifically, the memory cost of $A_2$ relative to $A_1$ was over 7 times larger whereas that of $A_3$ was over 20 times larger. Similarly, the power cost of $A_2$ was 16 times higher whereas that of $A_3$ was over 63 times greater in comparison to $A_1$. Thus, the above example serve to demonstrate that existing adaptation approaches do not scale well to larger and more complex applications.

Table 5-1 Application Statistics in Evaluation Scenario

<table>
<thead>
<tr>
<th>Application</th>
<th>Classes</th>
<th>Number of Class Couplings</th>
<th>Average Coupling per class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>A2</td>
<td>30</td>
<td>93</td>
<td>6</td>
</tr>
<tr>
<td>A3</td>
<td>94</td>
<td>299</td>
<td>4</td>
</tr>
</tbody>
</table>

However, the efficiency limitations of adaptive decision computation are not constrained to the object topology computation process alone. Often there are incumbent overheads associated in maintaining and updating information about application abstraction elements throughout the collaborating devices, which are discussed in more detail in section 5.2.3. In the above example, these overheads result in additional resource costs proportional to the increase in application size. In addition, these overheads are also influenced by an increase in the size of the collaboration and the degree of dynamism in the environment (resource availability fluctuation) as well as the duration of execution of the application. This makes existing adaptation approaches unscaleable to a number of factors (application size and complexity, number of collaborating devices etc., discussed in more detail in section 5.2.2.3), which are of crucial importance in adapting heavy applications in resource constrained heterogeneous environments. This is because adaptation decisions are typically computed by constrained devices (Rossi P. and Ryan C., 2005) and may occur frequently in dynamic environments such as mobile spaces in which users join and leave the collaboration and network disruptions are commonplace. In concert, these overheads greatly impact the overall utility of adaptation and
potentially marginalise or outweigh the gains that could be obtained from such adaptation.

Hence this chapter focuses on addressing these limitations by specifically targeting the overheads associated with maintaining and updating application abstraction elements as well as the cost of computing adaptation decisions as is discussed in more detail in sections 5.3.1 respectively.

5.2 Existing Work

Integral to any offloading strategy is an adaptive decision making algorithm that computes the placement of application components to remote devices based on changing context such as application usage and resource availability. In general, this process has two sub processes:

1) Construction and maintenance of an application model which abstracts the structure and behaviour of an application as discussed in Chapter 4.

2) Computation of adaptation decisions based on this application model.

As was discussed in Chapters 2 and 3, adaptive decision computation approaches can be classified into Non-Graph based approaches (Rossi P. and Ryan C., 2005; Ryan C. and Rossi P., 2005; Ryan C. and Westhorpe C., 2004) (object level approaches) and Graph based adaptation strategies (Gu et al., 2003; Ou et al., 2006; Ou et al., 2007b) based on the first sub-process discussed above. The following two subsections discuss each approach by focusing on efficiency and scalability aspects of their underlying processes.

5.2.1 Non-Graph Based Approaches

As discussed in Chapter 2-4, in non-graph based adaptation approaches, the decision to place an object to a given device is performed at object level granularity and thus an abstraction of the resource usage of each object is maintained for decision computation. As discussed in Chapter 2, depending on the site in which adaptation decisions are computed non-graph based approaches can be classified as either Global or Local adaptation.

Global Adaptation: In Global or Centralized Adaptation, a single dedicated and unconstrained machine performs adaptation decision-making. Other nodes within the collaboration periodically communicate their environment and software metrics (Gani H. et al., 2006) (discussed in Chapters 2 and 3) to this central node. Thus, the metrics
COMMUNICATED by a node includes the resource usage measurements of the device and
the individual objects in its address space. When an adaptation decision is required,
the central node is responsible for computing and effecting an adaptation decision.
For example, if a node within the collaboration runs out of resources the central node
computes an adaptation decision that offloads computation from the constrained
device to one or more resource copious devices within the collaboration. The object
topology is computed with the objective of providing optimum load balance and
performance improvement while minimizing inter-object network communication.
While Global Adaptation allows for the near optimal placements of objects-to-nodes,
the computation costs of computing an object topology (based on the approach
proposed in (Ryan C. and Rossi P., 2005)) for even a simple scenario with a few
objects in small collaborations can be prohibitively expensive (Ryan C. and Rossi P.,
2005). While Ryan and Rossi (Ryan C. and Rossi P., 2005) discuss the possible use
of Genetic Algorithms as a solution to reduce this cost, the approach would still be
computationally expensive compared to a decentralized approach to decision making
(discussed in the next paragraph). Another disadvantage of Global Adaptation is the
need for a reliable and unconstrained node for decision-making, which presents a
central point of failure and limits its applicability within ad-hoc collaborations of
constrained devices.

Local Adaptation: In Local or Decentralized Adaptation, decision-making is
computed on individual nodes. Environmental metrics (discussed in Chapter 2 and 3)
of each node are periodically communicated to every other node within the
collaboration. Unlike Global Adaptation, the metrics propagation includes only the
collaboration’s resource availability and not the software metrics of objects (Gani H.
et al., 2006) (e.g. number of method invocations, method response times etc.). When
a node runs out of resources, it computes an adaptation decision based on the
information it maintains about the collaboration and the metrics of the objects in its
memory space. As the adaptation decisions are computed by considering only a
subset of the overall object interactions, the decisions made are not as optimal as the
centralized approach. However, such an approach removes the central point of
failure, and offers a more scalable approach to adaptation decision-making. Hence,
this form of adaptation has been more widely considered by existing literature (Gani,
2010; Rossi P. and Ryan C., 2005).

However as discussed in Chapters 2 and 4, existing non-graph based adaptation
approaches (object-level approaches) omit coupling information from the decision
making process. This omission considerably limits adaptation efficacy and increases
the overall cost of these approaches as was demonstrated in Chapter 4, consequently making them infeasible for adapting heavy applications in constrained heterogeneous environments. As a result, these works are not used as a comparative baseline in this Chapter. However, the two distinct strategies for computing adaptation discussed above are applicable to the focus of this Chapter as discussed in section 5.3

5.2.2 Graph Based Approaches

As was established in Chapters 2 and 4, the current state-of-the-art (Gu et al., 2003; Ou et al., 2006; Ou et al., 2007b; Xiaohui Gu et al., 2004) approach to computing adaptive offloading decisions involves modelling an application as a dynamic cost graph. In this graph, vertices model application components (at class level granularity) with the weight of a vertex being the composite resource usage (memory, power, CPU etc.) (Ou et al., 2006) of all instances of a class, and edges represent the coupling amongst them where the weight of an edge is determined by invocation rate and invocation cost, again by aggregating data from all class instances. In this approach, computing the distribution of components to devices involves partitioning this application graph into a number of disjoint subsets/partitions where each partition identifies the list of classes (and their instances) that are to be offloaded to a single target.

The following subsection provides a brief background on graph partitioning approaches, so as to understand the basis and rationale of the techniques adopted in existing adaptive decision computation approaches while identifying some of the efficiency and scalability limitations inherent in the approaches. Furthermore, this requisite background facilitates understanding of the approach proposed in section 5.3.2.

5.2.2.1 Graph Partitioning, A Primer

Given a graph $G$, with a vertex set $V$ and an Edge Set $E$, the problem of partitioning this graph into $k$ disjoint subsets of equal (or bounded) sizes with minimal number (or weight) of edges connecting them is defined as Graph Partitioning. Finding an optimal solution to the problem is shown to be NP-Complete (Garey and Johnson, 1979) and hence a number of heuristics have been proposed as a result of the diverse domains for applications of this problem, which includes integrated circuit design (Kernighan and Lin, 1970) (Mead and Conway, 1980), task scheduling in multi-processor systems etc.
In general, graph-partitioning heuristics are classified as being either of Geometric or Non-Geometric types (Elsner, 1997). In the former, vertices are placed in a coordinate space that maps the physical (or logical) location of the entity being abstracted where the length of the edges connecting these vertices models the spatial proximity between them. These approaches are generally applicable to domains where the layout and connection of points are naturally modelled in coordinate space, for instance solving for map coordinates etc. In contrast, non-geometric heuristics are applicable to a wider range of domains and employ combinatorial or algebraic solutions to the graph-partitioning problem.

Graph partitioning algorithms can either be local or global. Local graph partitioning algorithms are used to make refinements to existing partitioning decisions and thus focus on a small section of the graph to improve partitioning. An example is the Kerninghan and Lin refinement algorithm (Kernighan and Lin, 1970), which aims to improve the partitioning decision by swapping boundary vertices between two partitions depending on the gains (edge-cut reduction) that could be achieved in doing so. On the other hand global approaches for partitioning graphs focus on the entire graph for computing partitioning decisions. Two popular approaches are Spectral Bisection (Hendrickson and Leland, 1995) (and its variations) and Multilevel partitioning (Karypis and Kumar, 1995; Karypis and Kumar, 1996a, b).

Multilevel Graph partitioning (Karypis and Kumar, 1995; Karypis and Kumar, 1996a, b) is the current state-of-the-art approach for computing partitioning decisions. The approach involves coarsening a graph down to a smaller size by first computing a maximal matching (Karypis and Kumar, 1995) of a graph. The process involves randomly selecting a vertex and matching it with an unmatched neighbouring vertex which is selected based on one of a number of criteria (Karypis and Kumar, 1995). These matching criteria include Heavy Edge Matching, Heavy Vertex Matching, and Light Vertex Matching. For example, in the case of the more popular Heavy Edge Matching approach for instance, the most tightly connected neighbour is selected, whereas with Random Matching, a randomly selected unmatched neighbour is selected.

These matched vertices are then merged together and the process is repeated until a small graph size is obtained. Once the graph is small enough, a different graph partitioning algorithm (Hendrickson and Leland, 1995) is used, including less efficient but greater quality strategies (spectral bisection (Hendrickson and Leland, 1995) and inertial bisection (Williams, 1991)). Once the graph has been partitioned it
is successively expanded into the original graph (the right-hand portion of Figure 5-1 below). During this expansion phase various approaches could be used to refine the decisions performed by the coarsening phase by applying local refinement algorithms such as the Kernighan and Lin refinement (Kernighan and Lin, 1970), in which boundary vertices are swapped to improve the outcomes of the partitions. This can be performed on each refinement phase, as a greater degree of freedom (more vertices to swap) is introduced in each expansion phase of the graph. Figure 5-1 below demonstrates this iterative cycle approach of coarsening the graph in the left half of the picture.

![Multilevel graph partitioning approach (Elsner, 1997)](image)

**Figure 5-1 Multilevel graph partitioning approach (Elsner, 1997)**

### 5.2.2.2 Computing Application Partitioning Decisions

At a minimum, each partition must satisfy the constraint that the aggregate partition weight is less than or equal to the resource availability of the designated device, thereby satisfying adaptation objectives 1-3 discussed in section 4.1 (meeting the resource constraints of application objects and collaborating devices). Furthermore, the weight of edges whose ends fall on different partitions, which determines the object topology network cost, should be minimized according to objective 4 in section 4.1.

As discussed in the previous section while various classical heuristics discussed above exist (Chinthapanti, 2004; Fjallstrom, 1998; Karypis and Kumar, 1995; Karypis and Kumar, 1996a, b; Raghavan and Garcia-Molina, 2003), they are not explicitly targeted at constrained mobile environments, and thus a number of heuristics designed explicitly to partition dynamic class graphs for adaptive offloading have been put forward. For example, Gu et al.(Gu et al., 2003) proposed an algorithm derived from the classical min-cut (Stoer and Wagner, 1997) (discussed in 2.3.3), for adapting between a constrained device and a dedicated surrogate (resource copious machine). More recently, Shumao et al. (Ou et al., 2006) proposed an alternative form of the multi-level graph partitioning heuristic discussed above in
section 5.2.2.1, to adapt across multiple constrained devices by successively coarsening an application graph. This was done by randomly selecting vertices and merging them with their lightest (low vertex weight) but highly coupled (high edge weight) neighbour, until the number of vertices was equivalent to the number of collaborating devices. Each vertex in the resultant coarse graph is then mapped to a device in the collaboration and represents a partition of the original graph. For efficiency reasons, the Shumao et al. (Ou et al., 2006) approach omits the refinement phase of the classic multilevel approach discussed in section 5.2.2.1.

As this approach was shown in (Ou et al., 2006) to provide both better performance and efficacious adaptation (reduced edge-cut) in comparison to the approach by Gu et al. (Gu et al., 2003), it is used as a comparative baseline for the graph partitioning approach proposed in this paper.

5.2.2.3 Efficiency and Scalability Limitations

With an $O(|V|^3)$ runtime complexity of the Shumao et al. (Ou et al., 2006) approach (where $|V|$ is the number of vertices), computing an adaptation decision is expensive and unscaleable to application graph size. This cost is compounded in mobile environments, not only by the constraint of devices but also by the frequency of decision making necessitated by execution in a dynamic environment.

To quantify this complexity, the authors evaluated the partitioning time for different sized application graphs executing on an HTC G1 Smartphone (with 528 MHz processor). Whereas a small application graph consisting of 8 vertices took only 27ms to partition, an application graph with 2,408 vertices took in excess of 30
minutes, thereby demonstrating that even the best of existing partitioning techniques
does not scale well on constrained devices.

In addition to partitioning overheads, existing approaches require each
collaborating device to maintain a copy of the application class graph (shown as (1)
and (2) in Figure 5-2 above) which incurs collaboration-wide memory cost of \( O(NM) \)
where \( N \) is the number of devices and \( M \) is the application graph size. Equation (1)
and (2) below model the estimated memory utilization \((mu)\) of this approach on each
device and the entire collaboration respectively where \( N \) is number of devices and, \( V \)
and \( E \) represent the vertex and edge set in the application graph \( G(V, E) \). It is shown
that the memory utilization of the Graph \( G \) at a given device \( i \) denoted as \( mu_i(G) \)
is the sum of the memory utilization of both the vertices \((mu(v))\) and edges \((mu(e))\) of
the application class graph. In addition, the total memory utilization within the
collaboration \( mu_{total}(G) \) is the sum of all the memory consumed by the application
on each device. This derived model is used in the evaluations in section 5.4 to
illustrate the cost of this approach under varying environmental conditions defined by
these variables as compared to the proposed solution discussed in section 5.3.

\[
mu_i(G) = (mu(v) \times |V|) + (mu(e) \times |E|) \quad \text{where } v \in V \text{ and } e \in E \text{ for } G(V, E) \text{ and } i \in N \tag{1}
\]

\[
mu_{total}(G) = mu_i(G) \times N, \text{ where } N = \text{number of devices} \tag{2}
\]

Furthermore, since any device within a collaboration might need to compute an
adaptation decision (similar to local adaptation approaches discussed in section 5.2.1)
and each device only monitors the resource usage of its local components, it must
rely on frequent updates about changes to remote components from other devices,
which occurs either periodically or based on the degree of resource fluctuation in the
environment (shown as (3) in Figure 5-2 above). However, this process incurs power,
performance and network overheads, which are exacerbated by an increase in the size
of the adapting application, the number of collaborating devices or the degree of
resource or user dynamism in the environment.

As lead author of (Abebe and Ryan, 2011b), the author of this thesis derives a
predictive model of the update cost in terms of network utilization of graph based
approaches as shown in Equation (3). The equation quantifies the potential graph
update cost for an evolving graph \( G(V_i, E_i) \), a changing collaboration size \((N_i)\) a
CHAPTER 5 Improving Efficiency and Scalability of Object Topology Computation

specified update frequency of adaptation \((\beta)\), and duration of execution \((T)\) where the serialized size of an update payload \((\text{Size of Serialized method Parameters, SSP})\) are known for graph elements. This derived model is used in the evaluations in section 5.4 to illustrate the cost of graph based approaches under varying environmental conditions defined by these variables as compared to the proposed solution discussed in section 5.3.

\[
n_{\text{utotal}}(G(V, E)) = \sum_{i=1}^{T \times f} \left( 2 \times (N_i \times \text{SSP}(v)) + (E_i \times \text{SSP}(e)) \right)
\]  

(3)

A further limitation of existing approaches is that each new adaptation decision ignores the current object topology, such that subsequent topologies may bear little resemblance to those that preceded them. Therefore, large migration costs (wherein many objects are migrated to many different hosts to represent the new topology) can potentially occur for only marginal gains in efficacy (i.e. reduced edge-cut). Note that the migration cost of a single class consists of the transfer of its class file (assuming it is not cached by the device’s JVM) and all its serialised instances. In summary existing approaches to adaptive offloading incur overheads from partitioning, storing and updating application graphs, while incurring migration costs resulting from the computation of divergent object topologies.

5.3 Proposed Approach

This section details a novel application graph representation in which devices maintain graph vertices only for components within their memory space, and abstraction vertices called \textit{cloud-vertices} representing components in remote devices (subsection 5.3.1). In addition, a graph partitioning heuristic is also proposed (section 5.3.2), which leverages the new graph representation to compute both efficient and efficacious decisions as shown in the evaluations in section 5.4.

Prior to offloading, existing approaches pre-suppose that the device has the resource capacity to load, start and briefly run the initial runtime of an application in order to produce an initial class graph \((G_{\text{initial}})\) as a starting point for offloading. Since, this may not be possible for applications with large initial footprints starting on constrained devices, alternatives include performing an initial partitioning based on a statically constructed application graph (e.g. through offline code profiling) (Li et al., 2001) or delegating application start-up and/or initial graph construction to a more
powerful surrogate. However, since further discussion of such approaches is not the focus of this work, this thesis assumes that prior to using the distributed graph approach; a single device is hosting a running application and already has a representative class graph that can be partitioned using any existing non distributed heuristic. To provide a comparison with the state-of-the-art, the approach by Shumao et al. (Ou et al., 2006) is used for the initial partitioning in this study.

5.3.1 Distributed Local Application Graph

After the initial partitioning and subsequent migration of components to their designated devices, each device constructs a distributed sub-graph of the initial class graph, consisting only of local components and abstraction elements (cloud vertices) for remote devices. Specifically in the new class graph on device \( n \in N \), (Graph \( G_n(V_n,E_n) \) from expression (4)) the vertex set \( V_n \) is made up of two distinct types of vertices, local-vertices \( (V_{n}^{\text{local}}) \) and cloud-vertices \( (V_{n}^{\text{cloud}}) \); where any localVERTEX \( v_n \in V_{n}^{\text{local}} \) represents a class (and its instances) residing on the local machine \( n \), and any cloud-vertex \( (v_i \in V_{n}^{\text{cloud}}) \) represents all the components in some remote device \( i \). Similarly the edge set \( E_n \) is made up of a set of local-edges \( E_{n}^{\text{local}} \) connecting any two local-vertices, and cloud-edges \( E_{n}^{\text{cloud}} \) which connect a local-vertex with a cloud-vertex.

\[
G_n(V_n,E_n) \rightarrow V_n = V_{n}^{\text{local}} \cup V_{n}^{\text{cloud}} \quad \text{and} \quad E_n = E_{n}^{\text{local}} \cup E_{n}^{\text{cloud}} \tag{4}
\]

\[
G_{\text{global}}(V_{\text{global}},E_{\text{global}}) \rightarrow V_{\text{global}} = \bigcup_{n=1}^{N} V_n \quad \text{and} \quad E_{\text{global}} = \bigcup_{n=1}^{N} E_n \tag{5}
\]

\[
G_{\text{global}} \setminus G_{\text{initial}} = \{V_{\text{cloud}}, E_{\text{cloud}}\} \tag{6}
\]

Expression (5) describes the graph \( G_{\text{global}} \) stored by the set of \( N \) collaborating devices, which although disconnected can be considered an abstract distributed view of the application graph. Expression (6) states the overhead in terms of additional elements (cloud-elements) when comparing \( G_{\text{global}} \) to \( G_{\text{initial}} \) (as a set-theoretic difference). Note that these overheads are shown to outweigh the costs of storing the complete class graph on each device in section 5.4. The approach is illustrated in Figure 5-3 where label (1) shows the initial application graph executing on a given mobile device and label (2) shows the distributed application graph maintained by each device \( n_i \) wherein each cloud vertex \( n_i^{\text{cloud}} \) abstracts the components on a remote
device $j$ (label 3) and cloud edges connected to this vertex abstract the cumulative coupling of the local components in $n_i$ with components hosted in $n_j$.

Figure 5-3 Distributed application graph on each device after an initial adaptation decision

The process of generating the local graph $G_i$ from the initial graph $G_{initial}$ has four steps where each step involves the generation of all instances of a specific type of graph element (local vertex, local edge, cloud vertex, cloud edge). In the algorithm in Figure 5-4 the local and cloud vertices are created before the local and cloud edges, however for clarity the local edges and vertices are discussed first followed by their cloud counterparts.

1) local-vertices: The algorithm iterates over the existing vertex set (lines 4-11) to identify classes which are hosted locally, and inserts a corresponding new vertex with identical properties into the local graph $G_i$ as shown in lines 5-7.

2) local-edges: The algorithm iterates over the existing edge set (lines 13-23) to identify edges for which both end points are hosted locally, and inserts a corresponding new edge with identical properties into the local graph $G_i$ as shown in lines 14-18.

3) cloud-vertices: For local vertices which have remote endpoints, a weightless vertex called a cloud-vertex is created to abstract the device hosting the remote component (lines 7-10 and 26-35). While it would be intuitive to set the weight of a cloud-vertex as the cumulative weight of all the remote components on the abstracted device, a network overhead would be required to keep the value current, and since not required by the partitioning algorithm proposed in section 5.3.2 is omitted.
4) cloud-edges: For edges which have a local and a remote endpoint, an edge with the same weight called a cloud-edge is created which connects the local vertex (step 1) to the cloud-vertex (step 3) representing the device hosting the remote class. If a local component is connected to more than one component placed on the same remote device, a single cloud-edge, which aggregates the weights of all such connections, is maintained as shown in lines 18-22 and 37-43. Unlike a cloud-vertex the weight of a cloud-edge must be maintained in order to perform effective partitioning, however since this can be collected by monitoring incoming and outgoing remote calls (from which edge weights are inferred), there is no additional network overhead.

```java
FUNCTION buildDistGraph(Graph classGraph, Device localDevice)
    DistributedLocalGraph g
    Map<Device, CloudVertex> cloudVerticesMap
    FOR EACH v in V:
        IF v.isMappedTo(localDevice) THEN
            g.addVertex(v)
        ELSE
            CloudVertex cloudV = getOrCreateCloudVFor(v.mappedDevice)
            g.addVertex(cloudV)
        END IF
    END FOR
    FOR EACH edge e in E:
        IF e.end1.isMappedToDevice(localDevice) && e.end2.isMappedToDevice(localDevice) THEN
            g.addEdge(new Edge(g.getVertex(e.end1), g.getVertex(e.end2), e.weight))
        ELSE
            CloudEdge cloudEdge = getOrCreateCloudE(e.end1, e.end2, g);
            cloudEdge.addWeight(e.weight);
            g.addEdge(cloudEdge);
        END IF
    END FOR
END FUNCTION
```

```java
FUNCTION getOrCreateCloudVFor(Device remoteDevice, DistributedGraph g)
    CloudVertex cloudVertex = cloudVerticesMap.get(remoteDevice)
    IF cloudVertex is NULL THEN
        cloudVertex = new CloudVertex(remoteDevice.name, 0.0)
        cloudVerticesMap.add(cloudV);
    END IF
    RETURN cloudVertex;
END FUNCTION
```

```java
FUNCTION getOrCreateCloudE(Vertex end1, Vertex end2, DistributedGraph g)
    Edge e = g.getEdgeBetween(g.getVertex(end1), g.getVertex(end2))
    IF e is NULL THEN
        e = new CloudEdge(g.getVertex(end1), g.getVertex(end2))
    END IF
    RETURN e;
END FUNCTION
```
5.3.2 Partitioning of a distributed local application graph

To evaluate the efficiency and efficacy of the distributed abstract graph representation described above, the author of this thesis derives a multilevel graph-partitioning algorithm to meet the specific needs of the new representation. Since our modification to the classic multilevel heuristic (Karypis and Kumar, 1995) are partly based on the work of Shumao et al. (Ou et al., 2006), similarities are explicitly acknowledged where relevant below.

Adaptation first requires the identification of a list of candidate devices which can offer resources (discussed in Chapter 3) as well as devices which were once candidates and are hosting application components but are no longer candidates for further adaptation (non-candidate nodes). Using the resource availability of these devices as inputs, the partitioning proceeds in four steps as follows:

```
FUNCTION computePartition(DistributedGraph g, Device localDevice, List<Device> candidates)
    List<Partition> partitions
    // create separate partition for anchor vertices
    partitions.add(new Partition(localDevice))
    FOR Vertex v: g.getVertices:
        IF v.isAnchorVertex THEN
            // merge anchor vertex to local device partition
            partitions.get(0).merge(v)
        ELSE IF v.isCloudVertex
            IF candidates.contains(v.mappedDevice)
                // create a separate partition for each cloud-vertex of a candidate device
                partitions.add(new Partition(v, candidates.get(v.mappedDevice)))
            ELSE
                // merge cloud-vertices of non-candidate devices to anchor
                partitions.get(0).merge(v)
            END IF
        END IF
    END FOR
    FOR Device dev : candidates
        IF !g.hasCloudVertexFor(dev) THEN
            v = new CloudVertex(dev);
            newPartition = partitions.add(new Partition(v, dev));
            newPartition.merge(g.getHeaviestFittingUnmatchedVx(dev))
        END IF
    END FOR
    WHILE(|V| > candidates.size + 1)
        boolean success = performMaximalMatching(g);
        IF !success THEN
            return false; // partition failure
    END WHILE
```
1. Merging of Anchor Vertices

Firstly, anchor vertices, which represent system classes or classes that cannot be offloaded from the device because they either access local resources (data sources, GPS, Camera etc.) or make device specific calls, are progressively merged into a single partition that will not be migrated. A merge combines two vertices where the resource usage and coupling pattern of the new vertex is the aggregate of its constituents. In this case, the aggregate weight has an upper bound taking into account the load mitigation requirements of the adapting device. For instance if the device wanted to reduce its load (memory, power consumption, CPU etc.) by 60%, the total weight of the anchor vertices merged to this partition must be within 40% of
the total components hosted on the device, otherwise the partition is considered to have failed.

As shown in the algorithm in Figure 5-5, merging proceeds with the algorithm iterating over the vertex set (lines 6-21), while identifying and merging classes which are annotated as being tied to the device, with the newly created local device’s partition (line 7-10). Figure 5-6 on page 121 shows the merging of 4 anchor vertices (stationary classes) on a distributed application graph hosted on an HTC G1 Smartphone adapting the NASA-WWJ (National Aeronautics and Space Administration, 2004) application between an Amazon EC2 instance and an HTC IPAQ device (the scenario is discussed further in the evaluation in section 5.4).

2. **Cloud-Vertices for candidate devices**

   For each candidate device, the corresponding cloud-vertex in the local graph is added to a new partition whose constraints represent the amount of resources the device can offer (lines 11-15 in Figure 5-5). For instance if a remote device \( n \) can offer the adapting device 100MB of memory, then its corresponding cloud vertex \( v^\text{cloud}_n \) in the local graph is attached to a new partition whose constituent classes (when they are later selected) cannot exceed this amount, thus ensuring remote devices are not overloaded. Note that since a cloud-vertex itself is weightless it does not contribute to this limit. If a candidate device does not have a cloud-vertex in the local graph because it is either new to the collaboration or does not currently host components that are coupled to the local device, a new cloud-vertex is added to the graph and assigned its own partition (lines 23-29). However, since this cloud-vertex would not have an edge linking it to other components in the local graph, the heaviest non-anchor local vertex that can be accommodated by the new device is merged to the partition.

3. **Cloud-vertices for non-candidate devices**

   Cloud-vertices representing non-candidate devices are merged to the anchor partition created in step 1 ensuring that the components in these remote devices are not selected for offloading to other devices (lines 15-19). Consequently, in conjunction with step 2, adaptation decisions can implicitly account for existing topology insofar as adaptation is limited to offloading local components to candidate devices, thereby reducing migration cost as discussed in section 5.4.3. Figure 5-6 on page 121 illustrates this process through the adaptation scenario discussed in step 1) above, wherein only the Amazon EC2 instance is a candidate device.
4. **Coarsening Phase**

Finally, the graph is progressively coarsened (lines 31-38 and 44-71), by computing its maximal matching in which vertices are randomly selected and merged to their lightest (low vertex weight) but most coupled (heavy edge weight) neighbour. While this is similar to the approach by Shumao et al. (Ou et al., 2006) it is different in that heavy vertices need not be selected to serve as core partitions for remote target devices during each maximal matching as these are pre-fixed to cloud-vertices instead. The process also excludes the merge of any two cloud-vertices with each other or with the anchor partition from step 1).

![Figure 5-6 Partitioning Process when adapting between a mobile device and an Amazon EC2](image-url)
The graph coarsening (i.e. merging of vertices) repeats until all vertices have been merged to either the anchor partition (classes that will remain on the device) or candidate-device partitions. If intermediate vertices cannot be merged to any of the core partitions without violating their constraints, the partitioning is said to have failed and the process is halted. Figure 5-7 illustrates a possible partitioning outcome of the adaptation scenario discussed in step 1) and visualized in Figure 5-6 above. It can be seen that the classes to be offloaded to the Amazon EC2 are shown grouped with its cloud vertex whereas vertices remaining on the adapting HTC G1 smartphone are shown grouped with anchor vertices (stationary classes) and the non-candidate cloud vertex of the HTC IPAQ device.

5.3.3 Modifying a distributed graph to reflect topology changes

Assuming successful partitioning, classes are migrated to their designated remote devices with each device updating its local graph to include the newly arrived components. Specifically, the adaptation target(s) creates new 1) local-vertices for the newly arrived components 2) local-edges to represent the couplings of these components to other components on the device including the newly arrived components, 3) Cloud-vertices for devices that these new components have couplings to (if one does not already exist) and 4) cloud-edges for the coupling of these new components to their source device or other remote collaborating devices.
Similarly, the source device updates its local graph by 1) removing local-vertices of offloaded components 2) replacing edges from a local component to a migrated component with cloud-edges to the cloud-vertices of the target device(s). While the process of modifying an application graph after adaptation, introduces overheads which are not required in existing approaches, the performance benefits of removing the need to frequently update local application graphs (which is required by existing approaches) is shown to clearly outweigh these overheads as shown in the performance comparisons in section 5.4.

### 5.3.4 Adaptation by Delegation (Adaptive On-loading)

Although the principle function of the partitioning algorithm is to offload resources, if a client device needs to pull back remote components to improve utility, it can do so by informing a remote peer of the amount of resources it would like to accommodate. Hence, in such a case any device, \( n \), wishing to on-load components, would have to first select the remote target from which it would like to receive components, which can be done in one of two ways: 1) The device, \( n \) inspects its own distributed local graph to determine the remote device(s) to which its local components have highest coupling and then request that this device compute an offloading decision with it (\( n \)) as the sole candidate, or 2) The device \( n \), requests all other capable devices to compute a partitioning decision (between themselves and \( n \)) so as to determine which of these devices (and their partitioning decisions) can provide the greatest utility (lowest edge-cut etc.). The latter is expected to incur higher global (collaboration wide) computational cost, since it requires a number of devices to compute a graph partitioning decision (albeit on local sub graphs and not the global graph) and hence the former approach is explored further in the evaluation in section 5.3.4.

The process of adaptive on-loading is different from existing work in the following two ways 1) The delegated remote candidate device computes the adaptation as opposed to the source device in existing approaches, 2) The decision is computed on the distributed local graph of the remote adapting device (candidate device) as opposed to the global application graph in existing works. Since \( V_n \setminus V_n^{\text{cloud}} \subseteq V \) for an adapting device \( n \in N \) with a vertex set \( V_n \) and an application with a global vertex set \( V \), then \( O(|V_n|) = O(|V|) \), such that the approach would be less computationally expensive than existing work.
5.3.5 Discussion on Efficiency and Scalability

From the discussions in previous subsections, it is inferred that the proposed approach provides several advantages over existing (Non-Distributed) work. Firstly, it reduces the memory overhead of storing the application graph on each device. Equations (7) shows the expected memory usage of the distributed graph approach on a given device $i$ as expressed by its local vertex set ($V_i$), edge set ($E_i$), cloud vertices ($V_i^{\text{cloud}}$) and cloud edges ($E_i^{\text{cloud}}$). Similarly Equation (8) shows the collaboration wide memory usage of the proposed approach, which shows that unlike the existing approach (memory utilization shown in equation (1)), the distributed strategy requires additional elements to maintain remote device abstractions. However, it is expected that this overhead would be outweighed by the memory cost saved from reducing the size of the application graph stored on each device, as is later demonstrated in the evaluations in section 5.4.2.1.

$$
m_u(G) = (m_u(v) \times |V_i|) + (m_u(e) \times |E_i|) + (m_u(v^{\text{cloud}}) \times |V_i^{\text{cloud}}|) + (m_u(e^{\text{cloud}}) \times |E_i^{\text{cloud}}|)$$  \hspace{1cm} (7)$$

$$
m_u_{\text{total}}(G) = (m_u(v) \times |V|) + (m_u(e) \times |E|) + \sum_{i \in N} [(m_u(v^{\text{cloud}}) \times |V_i^{\text{cloud}}|) + (m_u(e^{\text{cloud}}) \times |E_i^{\text{cloud}}|)]$$  \hspace{1cm} (8)$$

Secondly, the approach removes the network, performance and power costs associated with graph updates, thus making the approach more efficient in mobile environments where bandwidth and battery power are often scarce or costly resources. Consequently, the approach is more scalable to application and collaboration sizes relative to existing approaches as is later quantified in the evaluations in section 5.4. Thirdly, the approach improves adaptation performance by reducing the overall graph size considered during adaptation. This is because the cost of computing a graph partitioning decision is dependent on the size of the application graph considered during decision making (as discussed in more detail later in section 5.4.2). Lastly, it is expected that the approach would reduce the migration cost by accounting for the existing topology, since adaptation is limited to offloading local components to candidate devices. These advantages are quantified in the following section.
5.4 Experiments and Results

An evaluation to compare memory, network, and power usage, as well as the performance and adaptation efficacy of both strategies was conducted. The evaluation utilized three heterogeneous devices and three open-source applications each of which had runtime resource utilization greater than the capacity of the mobile device. Since the focus of this paper is the comparison of adaptation strategies, additional comparisons against a no-adaptation case or thin-client operation were not conducted; since the utility of adaptation over these approaches has already been demonstrated previously (Gu et al., 2003; Ou et al., 2006; Ryan C. and Westhorpe C., 2004; Xiaohui Gu et al., 2004).

5.4.1 Evaluation Materials and Scenario

Test Applications: Since the focus of this work is on enabling adaptation of computationally heavy applications which might not inherently possess distributive capabilities, the following three open-source applications were considered for evaluation: 1) A Java based n-body simulator using the Barnes-Hut algorithm (Cahoon and McKinley, 2001); 2) a Hospital System Simulator (Cahoon and McKinley, 2001) and 3) the NASA World Wind Demo application (National Aeronautics and Space Administration, 2004). The runtime class graph of the Barnes-Hut implementation consisted of 8 classes and 15 edges, that of the Hospital Simulator consisted of 7 classes and 9 edges, whereas the NASA-WWJ application consisted of 80 classes connected with 197 edges. The respective class graphs of these applications excluded System classes (java.* etc.) which had their requirements and resource usage implicitly represented by application classes which utilized them. The GUI component classes and main thread entry-point class were declared as anchored classes (see section 1) for each application.

Collaborating Devices: To represent the potential diversity of devices within a pervasive collaboration environment, the following devices were used 1) an HTC Dream (G1) Smartphone with 528 MHz processor and 192MB RAM, running Android OS version 1.6, 2) An Amazon EC2 (Elastic Compute Cloud) Micro-Instance with 1.7GHz processor, 615MB RAM, running Windows Server 2008, and 3) an Intel Core 2 Duo 2.5GHz laptop with 4GB RAM running Windows 7 Ultimate. All devices were setup under laboratory conditions with non-essential services and applications halted or removed. The mobile device was connected to the laptop over
an IEEE 802.11g Wi-Fi connection and to the Amazon EC2 Instance over 3G (provided by the Australian Optus Network service, 2100 MHz).

**Adaptation Decision Computation Engine:** An adaptation decision engine was developed which included the following two subcomponents: 1) a light-weight and dynamic application monitoring sub-system, which constructs and maintains a runtime component graph reflecting the resource usage, performance and coupling information of components. This information is obtained by injecting resource monitoring capabilities into an application using Byte-Code Injection. 2) An adaptation sub-system which computes application graph partitions based on the algorithm proposed in Section 5.3. The adaptation engine should readily plug-in to object mobility frameworks such as (Fahringer, 2000; Ou et al., 2006; Ryan C. and Westhorpe C., 2004; Xiaohui Gu et al., 2004) and is available from the author upon request.

**Experiment Scenario:** The evaluation scenario considered both the case in which a device would adapt to alleviate constraint (adaptive offloading), and the case in which it would choose to on-load back resources once this constraint was alleviated (section 5.3.4). Specifically the scenario involved the HTC Smartphone progressively running out of resources and incrementally offloading 5% of an application to one of the remote target devices (Amazon EC2 or laptop) until either all offloadable components were migrated, or as much as 95% of the application had been offloaded. Once this stage was reached, the constraint of the device was gradually alleviated in 5% increments, triggering adaptive on-loading adaptations until the Smartphone had once again hosted as much as 95% of the application’s resource requirements.

The percentage of an application to be offloaded was determined by the extent to which memory usage of the application exceeded the memory constraint of the mobile device. During each adaptation, the remote device offered the adapting device only as much resources as it requested, for instance if during its 4th incremental adaptation the device sought to offload 20% of an application, the target device would offer only as much as 20% plus a small configurable amount for flexibility. In reality, the device’s gradual resource constraint could be caused by a heavy application with a growing resource usage, or a user progressively executing background tasks. Similarly, a scenario with resource-sparing remote devices instead of resource-copious servers was assumed to be more realistic since remote devices could be serving other adaptations or unrelated tasks in parallel; or in the case of a cloud computing device, additional computation could incur financial costs. In
addition, such a scenario better represents cases where other constrained mobile devices (peers) are used as remote adaptation targets.

![Figure 5-8 Distributed Graph maintained on a Smartphone during an adaptation of the NASA-WWJ application](image)

To better aid in visualizing the adaptation process, Figure 5-8 illustrates a snapshot of the distributed local graph maintained on the Smartphone when hosting approximately 90% of the NASA-WWJ application, while the Amazon EC2 instance hosted approximately 10%. The local-vertices are shown in orange whereas the remote components (components on the EC2) are represented by the single cloud-vertex shown in green. It can be seen that at this stage only a few nodes have been offloaded, and the cloud-edges to the EC2 are minimal in relation to the coupling of local-vertices inside the Smartphone. This changes as the adaptation proceeds and more components are subsequently offloaded from the Smartphone to the EC2 as shown in Figure 5-9, which shows a snapshot as the device hosting approximately 30% of the application. The figure shows that the number of local vertices has considerably reduced on the Smartphone, and that more cloud-edges are maintained with the EC2. However, this does not necessarily mean that a larger edge-cut would result as the weight and not number of edges is a measure of edge-cut, and more offloading adaptations means that local components which are more coupled to remote components (instead of local ones) are offloaded thus potentially reducing total edge-cut (the edge-cut results are discussed in section 5.4.3). Figure 5-10 shows the distributed graph maintained by the EC2 at the same instance of adaptation when it hosted approximately 70% of the application, which can be seen to maintain more
local-vertices (than the Smartphone) and a high number of cloud-edges to the Smartphone.

![Figure 5-9 Distributed Graph on Smartphone when hosting approximately 30% of the NASA-WWJ application](image)

5.4.2 Results

The proposed approach is compared to the existing non-distributed work by Shumao et al. (Ou et al., 2006) below. Comparison includes memory (section 5.4.2.1), network (section 5.4.2.2), performance (section 5.4.2.3) and power costs (section 5.4.2.4), as well as adaptation efficacy (section 5.4.3).


5.4.2.1 Memory Utilization

As discussed in sections 5.2.2.3 and 5.3.5 the memory cost of each adaptation algorithm is determined by the size of the runtime graph required for adaptation. While the existing approach requires the complete runtime graph to be stored on each device (section 5.2.2.3 equations (1) and (2)), the proposed strategy requires only local subsets of the graph to be maintained by a device (section 5.3.5 equations (7) and (8)) Nevertheless, the proposed approach requires additional memory for storing cloud-vertices and cloud-edges that is not required in the non-distributed approach. Figure 5-11 shows the memory consumption of each device during progressive adaptation (partitioning) of the NASA WWJ application for the proposed approach and that of a single device for the existing approach (since each device has the same memory footprint). The results are in-line with the predictive models from section 5.2.2.3 and 5.3.5, showing that while the memory utilization of the non-distributed approach remains approximately constant throughout the adaptation; that of the proposed approach decreases by an average of 62% for the adapting device, and 38% for the remote target.

Furthermore, the collaboration-wide memory overhead of the non-distributed approach was on average 99.5% greater than that of the proposed approach. The results for the other applications showed similar results with an average collaboration-wide memory overhead reduction of 37.2% for the Health Simulator, and 40.1% for the Barnes-Hut implementation when using the proposed approach as shown in Figure 5-12. In addition, Figure 5-13 illustrates the predicted collaboration wide memory utilization of each approach as collaboration and application sizes
increase during adaptation, showing that the proposed approach is more scalable than the non-distributed approach.

![Collaboration-Wide Memory Utilization of Distributed (Proposed) Approach Vs Non-Distributed (Existing) Approach](image1)

**Figure 5-12** Collaboration-Wide Memory Utilization of Distributed (Proposed) Approach Vs Non-Distributed (Existing) Approach

![Predicted Collaboration wide memory utilization of existing approach (top surface) Vs. proposed approach (lower surface) as device and application graph sizes increase](image2)

**Figure 5-13** Predicted Collaboration wide memory utilization of existing approach (top surface) Vs. proposed approach (lower surface) as device and application graph sizes increase

### 5.4.2.2 Network Utilization

The network cost of the existing non-distributed approach is the result of three factors: 1) sending and receiving graph updates, 2) migration of objects and classes based on an adaptation decision, and 3) communication between objects in different localities (edge-cut of an application partitioning decision). While the last two are common to both approaches the overheads of graph updates are not incurred by the proposed approach. This section quantifies 1) and 2) with the aid of Figure 5-14 and Figure 5-15, while 3), which is a measure of the efficacy of adaptation decisions computed (as discussed in Chapter 4) is discussed separately in section 5.4.3.
Figure 5-14 quantifies the graph update cost after a complete adaptation involving both the gradual offloading of components as the constraint on the Smartphone increased, and the later on-loading of components as this constraint was gradually alleviated. The figure shows that as much 909 KB for NASA-WWJ, 45.6 KB for Barnes-Hut and 38.6 KB for the Health Application were transmitted between the adapting devices as a result of graph updates.

These are in fact conservative estimates since adaptation was performed between only two devices and graph updates were performed during adaptation iterations only, which presents the best case of the non-distributed algorithm. In practise, graph updates would need to occur more frequently to reflect changes in the resource usage of an application. Furthermore, collaboration size would typically be larger, thus incurring more cost than the best case scenario presented. Figure 5-15 provides a visualization of the expected network costs for adaptation, based on the predictive model in equation (3) in section 5.2.2.3, under increasing collaboration and application sizes, assuming a periodic update frequency of once per minute and an execution duration of 2 hours. Overheads increase with collaboration size, incurring as much as 6 GB of network cost in a 100 device collaboration adapting an application with a 1000 element (total vertices and edges) graph. This shows the existing non-distributed approach to be less scalable to increasing collaboration and application sizes, as well as dynamism of application behaviour.
Object Migration costs are incurred when transferring objects to their new localities based on an adaptation decision. Higher migration costs indicate that an adaptation decision resulted in a more divergent object topology than a previous distribution, thus requiring more object migrations. The proposed approach, whose adaptations are guided by the existing object-topology, was able to reduce migration costs by 12.2% in the case of the Barnes-Hut simulator (Figure 5-17), and 34.3% in the case of the NASA-WWJ application (Figure 5-16). However, due to the smaller class graph of the Hospital Simulator application, both algorithms resulted in the same object-topology when both completed successfully; however, 27% of adaptations failed using the existing approach (discussed in section 5.4.3).
5.4.2.3 Performance

The performance cost of computing an adaptation decision using the non-distributed algorithm consists of updating the local runtime graph and partitioning the complete application graph. The former, which is not applicable to the proposed approach, includes the network I/O delays of communicating graph updates prior to adaptation and the CPU cost of updating the local graph to reflect remote changes. As both factors are directly affected by the runtime graph size for a given application, the proposed approach provides performance benefits over the non-distributed approach by reducing the local graph size as components are offloaded. However, the performance results of computing the adaptive on-loading decisions once the Smartphone was fully constrained were excluded since, in the proposed approach, such an adaptation is performed by the delegated target device (discussed in section 5.3.4) instead of the device requiring the adaptation (Smartphone) as is the case in the existing approach. This means that the recorded performance would be that of different processor capabilities thus yielding uninformative results. Nevertheless, since the adaptive on-loading decision of the Smartphone would in fact be an adaptive offloading decision on the target device, the comparative results discussed below on the performance of adaptive offloading provide a comparison that encompasses this scenario.

Figure 5-18 shows the performance results of adapting the NASA WWJ application between the android phone and the laptop over a Wi-Fi network with an average latency of 10ms from which it can be seen that the adaptation decision computation time of the proposed approach decreases as more components are offloaded, with the worst case performance equivalent to that of the existing approach when the entire graph is considered during the first adaptation.

Figure 5-19 plots the adaptation computation times of the proposed approach against two separate cases for the non-distributed approach for each application; one case in which graph updates were omitted (so that graph partitioning performance can be compared directly) and a case in which they were included, which shows the total performance difference for each of the considered applications. In the former case, the proposed approach took on average 66% less computation time for NASA-WWJ on both target devices.
CHAPTER 5 Improving Efficiency and Scalability of Object Topology Computation

Figure 5-18 Adaptation decision computation time on HTC G1 of distributed approach vs. existing non-distributed (with and without graph updates)

When graph updates were factored in, this grew to 86% when adapting to the laptop and 98% to the Amazon EC2 instance, with a 3G average ping time of 230ms. The results for the Health Simulator and n-body simulation applications were similar with performance improvements of 19.5% and 16% respectively without graph updates and 85% and 88% when adapting to the laptop and 96% and 98% when adapting to the Amazon EC2 with graph updates.

In summary, the proposed approach provides improvements by removing the need for graph updates and reducing graph size required for adaptation, thus offering a more scalable adaptation approach to larger application and collaboration sizes.

Figure 5-19 Adaptation Performance Comparison Distributed vs. Non-Distributed
5.4.2.4 Power Consumption

The final power usage on the Smartphone after a complete adaptive offloading decision cycle had been performed was recorded for each approach and the results compared in Figure 5-20-Figure 5-22 below.

For the same reasons discussed in section 5.4.2.3 above, the power usage results of computing the adaptive on-loading decisions were excluded. Figure 5-22 shows that for the NASA-WWJ application, the proposed approach consumed 84% less power than the existing non-distributed algorithm when adapting to the laptop and 93% less power when adapting to the cloud machine as measured on the Smartphone\(^4\). Graph updates accounted for 32.2% of the total battery usage of the non-distributed approach when adapting over Wi-Fi and 69% when adapting over 3G to the cloud instance. A comparison which excluded the graph update costs of the non-distributed approach showed that the proposed approach still consumed 63% less power as a result of the reduced graph partitioning overheads discussed in section 5.3. Similar results were obtained for the Barnes-Hut and the Hospital Simulator applications with the proposed approach offering power cost reductions of 37% and 33% respectively when graph update costs were excluded, 89.7% and 90.2% when adapting to the laptop, and 96.9% and 97.1% when adapting to the cloud instance.

![Graph showing power usage comparison](image)

Figure 5-20 Hospital Simulator Adaptation power usage comparison

---

\(^4\)Power usage was measured using PowerTutor: http://ziyang.eecs.umich.edu/projects/powertutor/
The efficacy of an adaptation decision can be measured by the cost associated with the object topology it computes as discussed in Chapter 4. The inter-Object Communication of an object topology is quantified by the edge-cut of a partitioned application graph, which is the total weight of edges whose ends fall on different partitions. Higher edge-cuts imply that a number of highly coupled objects were placed in different partitions which in turn results in increased network and battery cost as well as reduced application performance caused by invocation delays. Figure 5-23 shows the edge-cut in terms of size of serialized parameters communicated amongst remote objects throughout the adaptation of the NASA-WWJ application, which included a phase of offloading as the device got constrained (positive percentage offload values in the X axis) and a case in which it computed on-loading decisions as its constraints were alleviated (negative percentage of offload values in the X axis).
X axis). For the NASA-WWJ application, the proposed approach resulted in an average edge-cut reduction of 62% when computing offloading decisions, and an average of 56% when computing the adaptive on-loading decisions, in comparison to the non-distributed approach. The reduction in edge-cut is attributed to the partitioning algorithm proposed in section 5.3.2 which ensures that vertices which are highly coupled to a remote candidate device are grouped with the corresponding cloud-vertex instead of the local device.

In the case of the n-body simulator and the Hospital Simulator applications, which had smaller graph sizes in comparison to the NASA-WWJ application, the computed edge-cuts were the same for both algorithms in cases where both algorithms computed partitions successfully. However, in contrast to the proposed approach, the non-distributed algorithm was unable to compute adaptation decisions for 6/36 of the n-body simulator adaptations and 10/36 of the Hospital Simulator adaptations because the approach could not successfully partition the application without violating constraints. While failure to achieve a specific imbalance is to be expected for graph partitioning heuristics (as discussed in Step 4 in section 5.3.2) the larger number of failures relative to the distributed approach is expected to be a result of the increased randomness in the selection of anchor points for adaptation in contrast to the proposed distributed approach in which anchor vertices are limited to the local (non-mobile object set) and cloud vertices for selected candidate devices. In addition the larger graph size of the non-distributed approach means that during the coarsening phase intermediate partitions (which do not belong to either anchor partitions) could form which are of a large sizes and thus unable to merge with anchor vertices without violating their constraints.

Figure 5-23 Edge-Cut comparison of Non-Distributed (existing) vs Distributed (Proposed) approaches, during adaptation of NASA World Wind Java Application
5.5 Conclusion

Existing approaches to adaptive offloading incur overheads from storing, updating and partitioning complete application graphs on each device, which limits their utility and scalability in resource constrained mobile and pervasive environments. Hence in order to address this aspect of adaptation as per research question B.1 in section 1.3.2, a novel distributed graph representation was proposed wherein devices maintain graph vertices only for components within their memory space, and abstraction vertices called cloud-vertices for components in remote devices. In addition, a novel graph partitioning heuristic was proposed. These approaches were shown to reduce network, power, and memory utilisation as well as the performance cost of adaptive offloading. Additionally, the efficacy of the generated partitions was also improved in terms of reduced remote object coupling, as well as reduced migration cost. This was demonstrated by a laboratory evaluation involving real-world open-source applications adapting on a Smartphone, a laptop, and an Amazon EC2 cloud compute instance. Thus, this Chapter addressed research questions B.1 (improving efficiency and scalability) identified in Chapter 1.
Chapter 6  Consolidated Adaptation Engine

In Chapters 3-5, the efficiency, scalability and efficacy limitations of existing adaptation approaches were addressed separately, by focusing on the different sub-processes of adaptation in isolation. However, as discussed in Chapters 1 and 2, improving the overall utility of Adaptive Computation Offloading in pervasive environments, requires the simultaneous improvement of the sub-processes of adaptation, across the same quality attributes of efficiency, scalability and efficacy, which in turn requires the integration of the approaches proposed in this thesis, into a single Consolidated Adaptation Engine.

This presents two main challenges: firstly the distributed candidate device selection approach proposed in Chapter 3, was discussed in the context of non-graph based adaptation, and hence a fitness computation strategy which accounts for the requirements of graph based approaches is requisite. Secondly, the efficacy of Hybrid Granularity must be balanced against the efficiency limitations that arise from its increased application graph size (as discussed in Chapter 4).

Therefore, this chapter is concerned with addressing these challenges to simultaneously improve the efficiency, scalability and efficacy of the overall adaptive decision computation process. The outcome is a Consolidated Adaptation Engine (CAE) that improves the overall applicability of Adaptive Computation Offloading in mobile and pervasive environments, as shown through an evaluation involving the adaptation of synthetic applications in a heterogeneous collaboration. It is shown that the proposed Consolidated Adaptation Engine improved adaptation memory utilization by as much as 65%; network and power utilization by as much as 99%, and performance by as much as 53%. Similarly, the approach resulted in efficacy improvements of up to 54% in comparison to an existing state-of-the-art approach.

The Chapter is organized into four subsections as follows: Firstly, section 6.1 briefly outlines some of the challenges in combining the approaches proposed in Chapters 3-5 and discusses a Consolidated Adaptation Engine, which addresses these challenges. Next, section 6.2 discusses the evaluation of the Consolidated Adaptation Engine in terms of the evaluation scenarios and settings considered. The results of the evaluation are presented and discussed in section 6.3. Finally, Section 6.4 provides a
discussion on the implications of the results in the context of the objectives of this thesis.

6.1 Consolidated Adaptation Engine (CAE)

As discussed above, the objective of a Consolidated Adaptation Engine is to simultaneously offer efficient, scalable and efficacious candidate device selection and object topology computation. The need for which is brought about by three limitations in using each optimization offered in Chapters 3-5. Firstly, the fitness score computation model of the candidate device selection strategy proposed in Chapter 3 was discussed in the context of non-graph based adaptation approaches; the inefficacies of which have been discussed in Chapters 4 and 5. Secondly, while the hybrid granularity approach discussed in Chapter 4 improved efficacy it yielded application graphs that were larger than their corresponding class graphs. This not only resulted in increased adaptation decision computation time, but would also require additional overheads for maintaining graph updates throughout the collaboration if applied to existing application graph management approaches. Lastly, while the distributed object topology approach resulted in improved efficiency and scalability, the efficacy gains were limited by the coarse-granularity at which this approach was used.

Hence, the seamless integration of the three approaches would involve the creation of an adaptation engine consisting of a distributed candidate device selection strategy and a distributed hybrid granularity application graph for computing object topology decisions. In this approach, devices need not communicate environmental or software metrics to the rest of the collaboration. Instead, when an adaptation decision is required, a request is multicast to the entire collaboration, at which point remote devices compute their fitness scores based on factors including their resource availability; the resource requirement of the adapting device, and the distributed application graph in their memory space as discussed in section 6.1.1. Furthermore, the distributed application graph maintained on each device is dynamically decomposed to improve efficacy of adaptation (Chapter 4). The decision to decompose a distributed graph is performed based on the size of the graph and the resource availability on the device so as to ensure that minimal resource costs are incurred as discussed in section 6.1.2. Since the granularity decomposition is performed on smaller sub-graphs instead of the complete application class graph, the resource costs are expected to be reduced as shown in the evaluation results in section 6.3.
Hence, the mechanics of combining the three adaptation optimizations proposed in this thesis must consider two primary factors 1) Fitness Score Computation for candidate device selection (discussed in section 6.1.1) and 2) Dynamic Hybrid Granularity Decomposition policy (discussed in section 6.2.2)

### 6.1.1 Fitness Score Computation

In the approach proposed in Chapter 3, each device must first compute its own fitness score, which determines its level of fitness to the adaptation request. The fitness score proposed in Chapter 3 factored in the resource load and resource availability on the candidate device relative to the request of the adapting source device. However, in the context of the distributed application graph strategy proposed in Chapter 5, this fitness score must also account for the degree of coupling with the adapting source device, so as to ensure adaptation decisions prioritize the reduction of existing high inter-device communication. Equation (1) below describes the degree of coupling between an adapting source device (\( n \)) and a candidate device (\( m \)), as the sum of the weight of the cloud edges linking local vertices on the candidate (\( m \)), with the cloud-vertex representing the source (\( u^m_n \))

\[
\text{coupling}_{SSP} = \sum_{\nu \in \text{Vert}_\text{local}} [e_{ssp}(\nu, u^m_n)] \text{ where } n, m \in N, u^m_n \in V_{\text{cloud}}
\]  

(1)

Alternatively, in the case of adaptive on-loading, the edge-cut reduction that can be offered by the candidate could be factored in, in place of the coupling degree between the source and the candidate (shown in Equation (3) below). The edge-cut is determined by computing a preliminary graph partitioning between the candidate device and the adapting source device.

Hence, equations (2) and (3) below, model the two alternative fitness score computation approaches. Equation (2) computes the score of a given device as the weighted power mean of the normalized score for its resource utilization (and availability) relative to the requirements of the source device and the degree of coupling of its local objects with this source. Similarly, equation (3) presents the fitness score computation model when the edge-cut reduction attainable is substituted for the coupling intensity between the devices. However, since computing a preliminary partitioning of the application graph on remote devices would result in additional computation cost (as discussed in section 5.4.3) only the first model (equation (2)) is used in the evaluation in section 6.2. Section 6.3.5 discusses the
efficacy of the candidate device selection process and the model presented in equation (2) as part of the overall efficacy of the adaptation decisions generated.

\[ S = (W_{ra}^R + W_{coupling}^R)^{1/r} \]  

\[ S = (W_{ra}^R + \left( \frac{W_{edge\_cut}}{I_{edge\_cut}} \right)^{1/r} \]  

6.1.2 Dynamic Hybrid Granularity Graph Decomposition

As discussed in Chapter 4, the Hybrid Granularity Graphs generated using the Class Graph Decomposition Strategy proposed in section 4.3 resulted in application graphs that were larger than their corresponding class graphs. While the approach offered increased efficacy that were in some instances orders of magnitude greater than their class graph counterparts, the approach incurred computational costs that negatively impacted the efficiency and scalability of the object topology computation process.

Hence, in order to leverage the efficacy of finer granularity while reducing the associated computational overheads, decomposition strategies are performed on smaller distributed local graphs created by the approach proposed in Chapter 5. Given that the decomposition of application graphs ensures that light vertices and edges are not fissured, the decomposition process does not affect the lightweight cloud vertices maintained by the distributed application graph.

Hence, in addition to the decomposition criteria listed in section 4.4.1, in the Consolidated Adaptation Engine, the decision to compute Graph Decomposition is additionally determined by the following factors:

1. **Minimum Ratio of Distributed-to-Class Graph Size:** In order to reduce the graph decomposition overheads, the granularity of an application graph is modified only when the local graph is determined to be small enough. This is assured by scheduling initial graph decomposition only when the Distributed Graph size \((d = |G_{distributed}|)\) to Class Graph size \((c = |G_{class}|)\) ratio falls below a pre-specified threshold \(T_{d/c}\).

2. **Bounded Hybrid Granularity Graph Size:** In section 4.4.1, criteria 3 and 4 limited the number of edges and vertices that resulted in the hybrid granularity graph. Similarly, the decomposition of a distributed application graph is bounded by the size of the original class graph so as to ensure that the computational cost.
on a single device does not exceed that of its corresponding class graph. This in turn distributes the cost of maintaining fissured graph elements throughout the collaboration, instead of incurring it solely on each device. The approach could be used in conjunction with the dynamic application graph granularity strategy discussed in section 4.6 wherein the granularity of an application graph could also be coarsened through contraction of the hybrid granularity graph. This would be beneficial if a device runs out of resources and would like to minimize the cost of the application graph maintained. In addition, the approach would also be beneficial in a scenario in which a device on-loads more fissured elements from remote devices and exceeds the initial class graph bound in its local memory space. However, in the evaluation presented in section 6.2, contraction of HGG is omitted for simplicity.

### 6.2 Evaluation

The objective of this section is to compare the efficiency and efficacy of an *Existing Adaptation Engine (EAE)* against two different forms of the *Consolidated Adaptation Engine (CAE)* as discussed below:

1. **Existing Adaptation Engine (EAE):** The adaptation engine against which the proposed consolidated engine is evaluated combines the state-of-the-art class graph based object topology computation approach (Ou et al., 2006) (discussed in section 5.2) with the candidate device selection strategy proposed by (Rossi P. and Ryan C., 2005) (discussed in section 3.1). Specifically, in this approach devices periodically communicate both software and environmental metrics and adaptation decisions are computed by partitioning application class graphs using the algorithm proposed in (Ou et al., 2006).

2. **Consolidated Adaptation Engine-A (CAE-A):** The first form of the proposed adaptation engine integrates distributed candidate device selection with a distributed object topology computation strategy. This form represents a scenario in which either granularity change is not required or is determined to be infeasible due to constraints discussed in section 4.4.1 and section 6.1.2. In CAE-A, when adaptation is required the adapting device multicasts a request to the collaboration. Capable remote devices compute a score based on the approach discussed in section 6.1.1 above (equations (1)-(3)), indicating their fitness level to the adaptation request. One or more of the fittest candidate devices respond to the adaptation request, after which the source computes a
partitioning decision on its distributed application graph, using the algorithm discussed in section 5.3.2. After each adaptation, the distributed application class graph maintained on the source and the target devices are updated to reflect the new topology.

3. Consolidated Adaptation Engine-B (CAE-B): The second form of the CAE integrates the granularity decomposition strategy discussed in section 6.1.2, with the adaptation engine discussed in CAE-A. Specifically, each device within a collaboration triggers granularity decomposition when conditions discussed in section 6.1.2 are satisfied on its distributed application graph. Adaptive object topology decisions are again performed on the hybrid granularity graph using the algorithm discussed in section 5.3.

6.2.1 Evaluation Materials

Experimental Applications: It was noted that in earlier Chapters, real-world applications were considered for evaluation in preference to synthetic cases. While this was performed in order to test the algorithms under more realistic execution requirements, it is noted that a small corpus of real-world applications is not representative of the diverse range of possible application behaviors. Hence, in this Chapter, synthetic applications with different behaviors (i.e. resource requirements, application graph sizes, coupling patterns etc.) are generated based on pre-specified constraints designed to explore diverse application behavior. These synthetic applications are created based on random graphs generated using two different graph generation models (JUNG, 2003) described below.

The first model involves a Probabilistic Model of Random Graph generation (Erdos and Renyi, 1959, 1960), wherein the connectivity of any two vertices, \(v, u\) is determined by some probability \(p\). Three applications A1, A3 and A5, of different sizes were generated using this model as shown in Table 6-1. A probability value of 0.2 was used for A1 and A3 and 0.1 for A5, so as to ensure graph-connectivity without resulting in the density of a complete graph (Weisstein) of the resulting application graph. The second model involved the Power Law Model for random graph (Eppstein and Wang, 2002) generation wherein the distribution of vertex degrees follows a power-law, with a small number of highly connected vertices. While some research into the existence of power law behavior in software coupling (as measured using different metrics) has been done (Baxter et al., 2006; Louridas et al., 2008), these were studied using a different definition of software coupling than
that adopted in this thesis (inter-class method invocation). However, it is noted that there is an intuitive relationship between these findings and the definition of coupling adopted in this thesis and as such this model generates more useful application graphs than other random cases. Hence three applications A2, A4 and A6 of different sizes were generated using this power law model (Eppstein and Wang, 2002) as shown in Table 6-1.

Table 6-1 Randomly Generated Application Graphs

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Vertices</th>
<th>Number of Edges</th>
<th>Average Degree</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>A2</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>Power law</td>
</tr>
<tr>
<td>A3</td>
<td>30</td>
<td>93</td>
<td>6</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>A4</td>
<td>47</td>
<td>100</td>
<td>4</td>
<td>Power law</td>
</tr>
<tr>
<td>A5</td>
<td>50</td>
<td>129</td>
<td>5</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>A6</td>
<td>94</td>
<td>299</td>
<td>4</td>
<td>Power law</td>
</tr>
</tbody>
</table>

The memory usages of the synthetic classes generated from the above graphs were designed to fall into one of three categories, Low, Medium and High, with each category defined as a specific memory utilization range. The distribution of memory utilization within all applications was kept constant and designed to yield fewer Low or High utilization classes and a larger number of Medium utilization classes, as shown in Table 6-2 below. The specific weights of classes within a given memory usage range were uniformly distributed.

Table 6-2 Generated Resource Utilization Specifications for Application Graphs

<table>
<thead>
<tr>
<th>Category</th>
<th>Memory Utilization Range</th>
<th>Percentage of Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>[100,1000]</td>
<td>20%</td>
</tr>
<tr>
<td>Medium</td>
<td>[1000,3000]</td>
<td>60%</td>
</tr>
<tr>
<td>High</td>
<td>[3000,5000]</td>
<td>20%</td>
</tr>
</tbody>
</table>

**Collaborating Devices:** To represent the resource constraint and potential diversity of devices within a pervasive environment, the following three mobile devices were used in the evaluation: 1) an HTC Dream (G1) Smartphone with 528 MHz processor and 192MB RAM, running Android OS version 2.2.1 2) An HTC HD2 Smartphone with 1GHz processor and 448MB RAM, running Android OS version 2.3.7, and 3) An Asus Transformer Prime Tablet with Quad-core 1.3GHZ processor and 1GB RAM, running Android OS version 4.0.3. All devices were setup under laboratory conditions with non-essential services and applications halted or removed. The devices were connected through an IEEE 802.11g Wi-Fi router.
Adaptation Decision Computation Engine: Both forms of the Consolidated Adaptation Engine were implemented in Java. The implemented adaptation engines consisted of the following six primary subsystems:

1) Environment monitoring subsystem: This subsystem is responsible for periodically polling the local device for resource changes (i.e. Memory Availability etc.) and triggering adaptation when the device exceeds a predefined resource load.

2) Candidate Device Selection subsystem: This subsystem is invoked when any device within the collaboration adapts and is responsible for: 1) computing and communicating adaptation requests when the local device adapts and 2) computing and communicating fitness score values when a remote device adapts. As discussed in section 6.1.1, this subsystem computes these values by querying the Environment Monitoring Subsystem (discussed above) and the Reflection Model Manager subsystem (see (3) below).

3) Application monitoring Subsystem: Is responsible for monitoring the resource usage of the executing application and relaying this information to the Reflection model manager (see (4) below). While the resource utilization of classes is measured periodically, inter-object method invocations are recorded throughout the execution of the application. The latter aspect is facilitated by injecting monitoring capabilities into the bytecode of the adapting application prior to execution. It is noted that a simple means of averaging the collected metrics through time is employed for the evaluations presented in this chapter instead of more sophisticated alternatives such as (Gani, 2010). This is because the accuracy and predictive properties of the collected metrics are not the focus of this work and have little bearing on the comparison of the proposed approach against the existing adaptation algorithms.

4) Reflection model manager subsystem: Reifies the software metrics collected by the Application Monitoring Subsystem for decision-making. Specifically, this subsystem handles the creation and management of the distributed application graph. As discussed in earlier chapters this application graph is dynamic and is hence updated as pertinent changes are relayed from the Application Monitoring Subsystem. These updates consist of the creation or removal of graph elements and the modification of their weights. In addition, the subsystem is also responsible for updating the distributed application graph as
adaptation decisions take place and new component topologies are brought into effect as discussed in section 5.3.

5) **Granularity Calibration Subsystem:** The subsystem reflects on the distributed graph maintained by the Reflection Model Manager to determine if granularity changes are required (as discussed in sections 4.4 and 6.1.2). In such an event, the subsystem works in coordination with the Reflection Model Manager to decompose the distributed application graph, based on the criteria discussed in section 6.1.2. In the evaluations provided in this chapter, this subsystem is invoked each time an adaptive offloading occurs.

6) **Topology computation subsystem:** computes adaptation decisions by partitioning the application graph stored by the Reflection Model Manager based on constraints offered by the Candidate Device selection systems.

The **Consolidated Adaptation Engine** should readily plug-in to object mobility frameworks such as (Fahringer, 2000; Ou et al., 2006; Ryan C. and Westhorpe C., 2004; Xiaohui Gu et al., 2004) and is available from the author upon request.

**Experimental Scenario:** For simplicity, the primary optimization objective for the purpose of the evaluation was memory utilization. However, as noted in Chapters 4 and 5, the approaches proposed in this thesis could be extended to include a range of different optimization objectives such as application performance improvement, power utilization reduction etc. The adaptation overheads incurred are expected to remain the same for different optimization objectives and hence the adaptation objective used does not have significant bearing on the comparison of the proposed adaptation approach against existing work.

The evaluation scenario involved the HTC Dream Smartphone progressively running out of resources and incrementally offloading 5% of each application (A2-A6) to one of the fittest remote target devices (HTC HD2 or Asus Tablet) until either all offloadable components were migrated or as much as 95% of the application had been offloaded. The percentage of an application to be offloaded was determined by the extent to which memory usage of the application exceeded the memory constraint of the mobile device. The scenario was run over a duration of 1 hour with a fixed software and environmental metrics update frequency of 0.5/minute being used (once every two minutes) in the case of EAE.

Additionally, in the case of the EAE, the fittest candidate was selected based on its resource availability and current load, whereas in both CAEs the approach proposed in section 6.1.1 was used. During each adaptation, the remote candidate
device offered the adapting device only as much resources as it requested. For instance, if during its fourth incremental adaptation the device sought to offload 20% of an application, the target device would offer only 20% plus a small configurable amount for flexibility. In reality, the device’s gradual resource constraint could be caused by a heavy application with growing resource usage, or a user progressively executing background tasks. Similarly, a scenario with resource-sparing remote devices instead of resource-copious devices was assumed to be more realistic since remote devices could be serving other adaptations or unrelated tasks in parallel. In addition, such a scenario better represents cases where other constrained mobile devices (peers) might be used as remote adaptation targets.

*Hybrid Granularity Integration:* In the case of CAE-B, an additional step of reconfiguring the granularity of the application was performed with the *Distributed-to-Class Graph ratio* (discussed in section 6.1.2) set to 0.75. The decomposition was performed in two steps, in order to highlight limitations of the granularity decomposition phase, with this cost included in the evaluation in section 6.3. Uniform constraints were used for the graph decomposition criteria discussed in Chapter 3, with the minimum value of a *High* resource utilization class (discussed under *Experimental Applications* above) used as a vertex weigh threshold $T_W$; the average degree of an application class graph used as the degree threshold $T_D$, and the initial class graph size used as the graph bound threshold ($T_{|V|}$ and $T_{|E|}$) as shown in Table 6-3. However, in the case of the larger application graphs, A5 and A6, higher vertex weight thresholds were used to prioritize fissuring of heavier vertices.

| Application | $T_W$ (Kilo Bytes) | $T_{degree}$ | $T_{sc}$ | $T_{|V|}$ | $T_{|E|}$ |
|-------------|-------------------|--------------|----------|----------|----------|
| A1          | 3                 | 2            | 100      | 5        | 5        |
| A2          | 3                 | 4            | 100      | 5        | 10       |
| A3          | 3                 | 6            | 100      | 30       | 93       |
| A4          | 3                 | 4            | 100      | 47       | 100      |
| A5          | 4                 | 5            | 100      | 50       | 129      |
| A6          | 4                 | 4            | 100      | 94       | 200      |

Table 6-4 Generated Hybrid Granularity Application Graphs

<table>
<thead>
<tr>
<th>Application</th>
<th>Fissured Vertices</th>
<th>Fissured Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A3</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>A4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>A5</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>A6</td>
<td>52</td>
<td>14</td>
</tr>
</tbody>
</table>
The Hybrid Granularity Graph generated introduced no new elements in the case of A1 and A2 as shown in Table 4. This is because the small number of heavy vertices in these graphs were offloaded prior to the decomposition of the application graph in both CAE approaches. On the other hand, it is observed that an application graph size increase of between 6% in the case of A4 and 25% in the case of A3 were recorded as shown in Table 6-4. The effects of the Hybrid Granularity Graphs generated for these applications are discussed in section 6.3 below.

6.3 Results

This section comparatively discusses the efficiency, scalability and efficacy of the CAE approach against that of the EAE, based on the results of executing the evaluation described in section 6.2. The section is organized into five subsections, with sections 6.3.1, 6.3.2 and 6.3.3 discussing memory, network and power utilizations respectively; followed by performance results in section 6.3.4 and efficacy results in section 6.3.5.

6.3.1 Memory Utilization

The section compares the memory utilization of the Existing Adaptation Engine (EAE) against the two Consolidated Adaptation Engines (CAE-A and CAE-B), discussed in section 6.1. The comparison excludes the memory cost of the underlying middleware, temporary memory resident data (e.g. Adaptation Requests and Fitness Responses), and the executing application, which are assumed the same for both approaches.

As shown in Equation (4) in the case of the existing adaptation engine (EAE), memory utilization of a device is a factor of two things: 1) The device metrics $e$ (such as the memory, network, power and CPU utilization) maintained about each device $n$ in the collaboration $N$, and 2) The complete Class Graph $G(V, E)$ of the executing application. The resulting collaboration-wide memory utilization of EAE is shown in Equation (5) as the sum total of the memory utilization of each device.

$$
u_{EAE}(V, E) = \sum_{e \in E} \nu(e)$$

$$\mu(\text{EAE}) = \mu(G(V, E)) + \sum_{m \in N} \mu(e_m)$$

(4)
Equation (4): Memory utilization of EAE on each device

\[ \text{mu}(EAE)_{\text{total}} = \sum_{n \in N} \text{mu}(EAE)_n \]  

Equation (5): Memory Utilization of EAE on collaboration environment

In contrast, the memory utilization of CAE-A on a given device \( n \) within the collaboration \( N \) is a factor of: 1) The Environmental Metrics \( e \) of the device itself, and 2) The distributed abstract class graph maintained for its local components as shown in equation (6). In addition to these costs, CAE-B includes the memory cost of fissured vertices and edges as shown in equation (7); the total memory cost of which is distributed across devices in the collaboration depending on the object topology in effect. The collaboration-wide memory utilization is shown in equation (8) as the sum total of the memory utilization of the CAE strategy employed on each device.

\[ \text{mu}(CAE_A)_n = \text{mu}(e_n) + (\text{mu}(v) \times |V_n|) + (\text{mu}(e) \times |E_n|) \]
\[ + (\text{mu}(v^{\text{cloud}}) \times |V_n^{\text{cloud}}|) + (\text{mu}(e^{\text{cloud}}) \times |E_n^{\text{cloud}}|) \]  

(6)

\[ \text{mu}(CAE_B)_n = \text{mu}(CAE_A)_n + (\text{mu}(v^{\text{fiss}}) \times |V_n^{\text{fiss}}|) + (\text{mu}(e^{\text{fiss}}) \times |E_n^{\text{fiss}}|) \]  

(7)

\[ \text{mu}(CAE)_{\text{total}} = \sum_{n \in N} \text{mu}(CAE)_n \]  

(8)

The evaluation results shown in Figures 1 and 2 are in line with the predictive models discussed above. The recorded memory utilization of the EAE was orders of magnitude greater than that of either CAE approaches in all applications. Specifically, the Consolidated Adaptation Engine-A (CAE-A) resulted in memory reductions of between 46% in the case of Application-A1 and 65% in the case of Application-A6. In the case of CAE-B, it was observed that in the applications where granularity modification had effect (Applications A3-A6 as shown in Table 6-4), the approach incurred more overhead in comparison to CAE-A but offered notably less memory utilization in comparison to the EAE, with memory utilization reductions ranging from 45% in the case of A3 to 63% in the case of A4.
6.3.2 Network Utilization

This section compares the network utilization of the proposed Consolidated Adaptation Engine against the Existing Adaptation Engine. Network utilization factors, which are common to both approaches, such as the external network utilization of the application (e.g., http requests etc.) and the inter-object invocation cost of objects in different localities, are omitted in this evaluation. The latter is discussed further as a measure of the efficacy of each adaptation engine decision in section 6.3.3.

Hence, from the discussion in Chapters 3 and 5, it is noted that the network utilization of the existing approach is a factor of the following two primary considerations: 1) Software Metrics Communication as effected through graph updates within the collaboration, and 2) Environmental Metrics Communication. These are expressed in equation (9) in terms of: the collaboration size ($N$); frequency of metrics updates ($f$), and the duration of application execution ($T$), wherein the frequency of update for both Software and Environmental metrics is assumed to be the same.

In contrast, in the case of the Consolidated Adaptation Engine CAE, the total network cost is a result of two factors which occur prior to each adaptation: 1) Communication of Adaptation Request Messages, and 2) Communication of Candidate Fitness values. Equation (10) expresses these costs in terms of the collaboration size $N$, Number of Adaptations $Na$ and the number of fit candidate devices $|C|$ that respond during each adaptation instance, $i$. 

Figure 6-1 Collaboration-wide Memory Utilization of EAE, CAE-A and CAE-B
\[ nu(EAE)_{total} = 2(N - 1) \sum_{i=1}^{T \times f} \left( N \left( \sum_{n \in N} nu(e_n) \right) + \left( |W_i| \times SSP(v') \right) + \left( |E_i| \times SSP(e) \right) \right) \quad (9) \]

\[ nu(CAE)_{total} = N(Na \times nu(R)) + \sum_{i=1}^{Na} [N(nu(M) \times |C_i|)] \quad (10) \]

The scenario described in section 6.2.1 results in a total of 30 metrics updates and 18 adaptations within a collaboration duration of 1 Hour. As discussed in Chapter 3, this represents a high adaptation count relative to the execution duration and is expected to be unlikely as the cost of such frequent adaptation could outweigh its gains. However, these values were used in order to provide a conservative estimate of the network utilization reductions offered by the proposed CAE relative to the EAE.

The evaluation results shown in Figure 6-2 were again in line with the expectations of the predictive models presented in Equations (9) and (10). It was observed that the proposed approach resulted in network utilization reduction in all cases relative to EAE. Specifically, in the case of the smaller applications A1 and A2, a reduction of 88% and 89% were recorded respectively, whereas in the case of the larger applications, which maintained more components and thus larger graph update costs, this difference was more prominent ranging between 97% in the case of A3 and 99% in the case of A6 as shown in Figure 6-2. The results for CAE-A and CAE-B were the same. This is because unlike the EAE, the decrease in application granularity, and the consequent increase in application graph size, does not incur additional network costs. Hence, the results not only assert that the consolidated adaptation approach is more efficient in terms of network utilization but also more scalable with regards to the application size.
6.3.3 Power Utilization

Power utilization is a result of a number of factors most of which are common to both approaches. This section is concerned with evaluating aspects that differentiate the two CAE approaches from the EAE. Specifically, in the case of EAE, power utilization is primarily a result of two factors: 1) Software and Environment Metrics Management and Communication within the collaboration, and 2) Computation of Object Topology decisions. In contrast, the power utilization of the Consolidated Adaptation Engine is a factor of: 1) Adaptation Requests and Fitness Responses Communication, 2) Computation of Object topology decisions, which is performed on different sized application graphs compared to that of EAE, and 3) Class Graph decomposition in the case of CAE-B.

To clearly identify the power cost of the different factors of both EAE and CAE approaches discussed above, the evaluation results are presented in three groups below. While the first two groups (Results 1 and 2 below) discuss the power consumption of the factors that occur pre-adaptation and during adaptation, separately, the third group discusses the total power utilization cost incurred on the adapting device. The groups are discussed in more detail below:

**Result 1:** presents the power utilized prior to the process of each object topology computation. This includes software and environment metrics collection and communication in the case of EAE (factor \( EAE \, I \) above) and Adaptation Request/Response communication in the case of CAE (factor \( CAE \, I \) above). Figure 6-3 and Figure 6-4 show that the proposed approach resulted in significant power
utilization reduction on both the adapting HTC Dream smartphone and the candidate device HTC HD2 with similar results recorded for the Tablet device. Specifically, in the case of the HTC Dream smartphone, power utilization reductions of between 95% and 96% were recorded for applications A1 and A2 and 99% for Applications A3-A6 as shown in Figure 6-3. Similarly, on the candidate Tablet device, power utilization reductions of 98% were recorded for A1 and A2 and 99% for applications A3-A6. In the case of the HTC HD2, power utilization reductions of between 85% in the case of A1 and 99% in the case of A2 were recorded as illustrated in Figure 6-4. Since graph updates do not occur in CAE, both CAE-A and CAE-B incurred the same power usage.

**Figure 6-3** Power utilization on adapting device (HTC Dream) resulting from Factor EAE 1) and CAE 1)

**Figure 6-4** Power utilization on candidate device (HTC HD2) resulting from Factor EAE 1) and CAE-1)

**Result 2:** Quantifies the power utilization of object topology computation and application graph decomposition (in the case of CAE-B), as recorded on the adapting HTC G1 Smartphone. Similar to **Result 1** it was observed that both the proposed
CHAPTER 6 Consolidated Adaptation Engine

CAE approaches were significantly more power efficient than the EAE as shown in Figure 6-5 and Figure 6-6. Specifically, CAE-A resulted in power utilization reductions of between 43% in the case of A4 and 60% in the case of A2. In comparison to CAE-A, CAE-B maintained larger application graphs and resulted in additional overheads from the process of graph decomposition, which resulted in modest overheads as shown in Figure 6-5 and Figure 6-6. However, the recorded power utilizations of CAE-B were notably less in relation to EAE, with power utilization reductions of between 25% (in the case of A1) and 38% (in the case of A5) in comparison to the Existing Adaptation Engine (EAE). In the case of A1 and A2 it is noted that while new-fissured elements did not arise from graph decomposition, the modest overheads relative to CAE-A were a result of computing the fissurability of classes.

![Figure 6-5 Decision Computation and Graph Decomposition Power usage (A1, A2)](image1)

![Figure 6-6 Adaptation Decision Computation and Graph Decomposition Power Utilization (A3-A6)](image2)
Result 3: The total power utilization incurred on the adapting HTC Dream Smartphone is illustrated in Figure 6-7. It is shown that the CAE approaches resulted in significant power utilization reductions in line with Results 1) and 2) above. Specifically, power cost reductions of between 91% in the case of A6 and 95% in the case of A2 were recorded for CAE-A whereas power utilization reduction of between 83% and 95% were recorded in the case of CAE-B relative to EAE. In summary, the results showed that the proposed Consolidated Adaptation Engine offered increased efficiency and scalability in terms of power utilization in contrast to the Existing Adaptation Engine.

6.3.4 Performance

The performance of each Adaptation Engine was measured by the total time spent computing an object topology decision as measured on the adapting device (HTC Dream Smartphone). This translated into the time taken to compute application graph partitioning decisions in both the EAE and CAE approaches. In addition, in the case of CAE-A and CAE-B, the time for selecting the fittest candidate device based on fitness report values returned was also included. While this cost would likely not be incurred in the delay-based report filtering approach proposed in Chapter 3, this was omitted for both simplicity and in order to conservatively estimate the network utilization reductions of the CAE approach in Section 6.3.2. Similarly, the latency cost of adaptation requests and reception of receiving fitness reports were again also omitted for simplicity and in order to clearly compare the decision computation times in isolation.
As shown in Figure 6-8 and Figure 6-9, the performance cost of the Consolidated Adaptation Engine was improved in comparison to the Existing Adaptation Engine for all but the smallest Application A1, for which the benefits of the distributed graph were outweighed by the performance cost of creating a distributed graph, selecting fittest candidate devices and in the case of CAE-B computing fissurability of graph elements. Similarly, in the second smallest application the performance improvements offered by CAE was marginal at 4.5% in the case of CAE-A.

However, for the remaining applications A3-A6, both CAE approaches resulted in performance improvements in comparison to EAE. Specifically, CAE-A resulted in adaptation performance improvements of between 46% in the case of A3 and 53% in the case of A6. Similarly, CAE-B resulted in performance improvements of
between 42% in the case of A2 and 54% in the case of A5. In addition, the performance difference between CAE-A and CAE-B were subtle despite the increased application graph size generated in CAE-B.

In fact, contrary to expectations, in the case of A4, CAE-B resulted in small improvements in adaptation performance relative to CAE-A. Further investigation of this result through the runtime visualization of the adaptation process and application graphs revealed an interesting additional benefit of the Hybrid Granularity Approach. While as expected, different adaptation decisions are computed by each approach as a result of the difference in granularity (as discussed in Chapter 4), it also resulted in different number of local components maintained on the adapting device at a given time. In the case of A4, this property resulted in the approach retaining fewer graph elements on the local device in order to better achieve its adaptation objectives, which consequently improved adaptation performance. While the performance improvement obtained by CAE-B relative to CAE-A for application A4 was subtle, the result highlighted a potentially interesting avenue for future work on how the granularity of an application could be calibrated not only to improve efficacy but also improve adaptation performance. The difference in adaptation decisions computed by the different approaches is highlighted by the efficacy results discussed further in section 6.3.5 below.

6.3.5 Efficacy

As discussed in Chapters 4 and 5, the efficacy of an adaptation approach is measured by the degree to which an object topology can satisfy adaptation objectives while minimizing the inter-device communication that results. As discussed in section 6.2 the adaptation objective in this evaluation is the load mitigation of the memory load on devices. Hence, in addition to achieving the required load mitigation on an adapting device (e.g. device is constrained and wants to offload 40% of its memory load etc.) the efficacy of the adaptation results are measured based on the degree to which the inter-device communication is minimized. It is noted that different adaptation objectives could have different efficacy metrics in addition to this in future, for instance the degree to which the application’s power utilization is minimized in the case of power optimization objectives.

As expected, the CAE-A approach resulted in efficacy improvements in line with the results in Chapter 5, with efficacy improvements ranging from 18% in the case of A3 to 54% in the case of A1 as shown in Figure 6-10. This efficacy gain is also a
result of the fitness computation model proposed in section 6.3.1, which allowed the selection of a more optimal target device for adaptation.

The introduction of hybrid granularity in CAE-B further improved the efficacy of adaptations in applications in which granularity change had effect (i.e. Application A3-A6 as discussed in section 6.2.1). Specifically, additional efficacy improvements ranging from 5% in the case of A6, to 21% in the case of A3, were obtained in comparison to the CAE-A approach.

In the case of Applications A1 and A2 (as noted earlier in section 6.2.1), no additional efficacy gains were recorded in CAE-B relative to CAE-A, since the graph decomposition resulted in no new elements being created. This is because the few heavy vertices, which would have been candidates for fissuring in these graphs, were offloaded prior to the decomposition of the application graph in both CAE approaches.

![Figure 6-10 Efficacy results of CAE-A, CAE-B and EAE](image)

### 6.4 Summary

The practicality of using Adaptive Computation Offloading as a means of enabling execution of Complex and Computationally Heavy applications in mobile environments requires simultaneous improvements in the efficiency, scalability and efficacy of various aspects of the adaptation process. This Chapter proposed a Consolidated Adaptation Engine that combined the optimizations to the candidate device selection and object topology computation sub processes of adaptation proposed in Chapters 3-5.
Through an evaluation conducted using heterogeneous mobile devices and synthetic applications, it was shown that the Consolidated Adaptation Engine offered improved Efficiency, Scalability and Efficacy of the overall adaptation process by reducing the network, memory, power and performance costs required for adaptation as compared to the state-of-the-art adaptation approach.

Hence, these approach proposed in this Chapter addresses the overall objective of this thesis, as outlined in questions 1-4 in Chapter 1.
Chapter 7  Summary and Conclusion

This chapter provides a conclusion to this thesis in three subsections as follows: section 7.1 first provides a summary of the approaches proposed in Chapters 3-6, from the context of the specific research goals outlined in Chapter 1. Then, section 7.2 discusses the practical applicability of the proposed adaptation engine to other computation offloading middleware. Finally, section 7.3 discusses future work relating to both the algorithms proposed in Chapters 3-6 and other aspects of adaptive offloading potentially related to the focus of this thesis.

7.1 Summary

The use of Adaptive Computation Offloading as a means for enabling the execution of computationally heavy applications in mobile environments requires improvements in the efficiency, scalability and efficacy of the adaptation process. This thesis addressed this problem by specifically focusing on the two sub-processes of adaptive decision computation: 1) Candidate Device Selection and 2) Object topology computation. To this end, three novel approaches for improving the efficiency, scalability and efficacy of each sub-process were first proposed in isolation and later combined into a consolidated adaptation engine, as discussed in more detail below.

Firstly, the candidate device selection sub-process was improved in Chapter 3, through a distributed approach for computing the fitness level of collaborating devices. The approach removed the need to communicate environment metrics, and allowed for the partial distribution of adaptation decision-making. An evaluation of the approach was conducted in two parts, with the efficiency and scalability aspects evaluated in Chapter 3 and the efficacy evaluated in Chapter 6. It was shown that in comparison to existing approaches, the distributed model was more resource efficient and more scalable with regards to the size of the collaboration and the degree of heterogeneity within the environment. Specifically, while for small collaborations, the existing algorithm offered up to 30% less network overhead, under medium to large-scale collaborations the proposed approach offered over 90% reduction in network consumption and as much as 96% reduction in power consumption. In addition, the approach maintained linear memory complexity in contrast to the quadratic complexity of an existing approach. Moreover, in conjunction with a graph-
based model for object topology computation, the approach was shown to yield more efficacious adaptation decisions in comparison to existing approaches.

Secondly, the efficacy of the object topology computation sub-process was improved in Chapters 4 through a novel hybrid granularity graph approach for computing adaptation decisions. The approach combined the efficiency of coarse-grained (class-level) adaptation with the efficacy of fine-grained (object-level) approaches. The approach was shown to improve adaptation efficacy by between 17% and 99% in comparison to class-level approaches, for the evaluated corpus of applications. While the approach was shown to be orders of magnitude more resource efficient in comparison to object-level approaches, it incurred modest overheads in comparison to class-level approaches as a result of the larger application graph maintained. This observation highlighted the limitations in efficiency of existing object topology decision computation algorithms and the lack of scalability in relation to the application size, complexity and granularity of adaptation.

Hence, the third aspect of this thesis focused on addressing the efficiency and scalability limitations of the object topology computation sub-process (Chapter 5). To this end, a novel distributed application class graph approach and an associated graph partitioning heuristic were proposed. The approach reduced the size of the application graph maintained and partitioned on each device and removed the need to communicate software metrics throughout the collaboration. The approach was shown to reduce resource utilization costs and improve adaptation decision computation performance in comparison to the existing state-of-the-art approach. Specifically, collaboration-wide memory reduction of between 37% and 50%; network consumption reduction of 100%; power consumption reduction of between 63% and 93%, and performance gains of between 19% and 93% were recorded.

Lastly, a Consolidated Adaptation Engine, which effectively integrated the approaches discussed above, was proposed in Chapter 6. To facilitate this integration, a new fitness score computation model that accounts for graph based adaptation, and a new criteria set for decomposing distributed class graphs (Chapter 5) were proposed. The Consolidated Adaptation Engine was shown to improve efficiency and efficacy in comparison to existing approaches and was more scalable with regards to the size of the application and collaboration considered. Specifically, the approach resulted in memory utilization reduction of as much as 65%, network and power utilization reduction of as much as 99% and performance improvement of as much as 53%. Similarly, the approach resulted in efficacy improvements in terms of edge-cut reduction of up to 54% in comparison to the existing state-of-the-art approach.
Hence, this thesis advances the state-of-the-art of adaptive computation offloading towards the rationale established in Chapter 1. Specifically, using the efficient, scalable and efficacious adaptation engine proposed (CAE) the seamless execution of computationally heavy desktop applications in pervasive environments is facilitated. This allows the realization of the scenario presented in section 1.1 wherein a user effortlessly executes a heavy desktop application (game) as he travels to work, by optimally utilizing the resources of nearby mobile devices.

### 7.2 Practical Applicability

The Consolidated Adaptation Engine discussed in Chapter 6, represents an integration of the work presented in this thesis, and is expected to be directly applicable to existing computation offloading middleware as follows:

**Abstraction and Generality:** As discussed in Chapters 4-6, the adaptation engine relies on the high-level architectural representation of the runtime behaviour of an application and is hence, largely agnostic to the implementation specifics of the adapting application. This makes it applicable to a broad range of existing desktop software and adaptable to related architectural domains (e.g. Service Oriented Architectures).

**Generality of Adaptation Objective:** The approaches presented in this thesis are not reliant on specific optimization objectives and can hence be extended, with minor modifications, to a wider range of offloading objectives, thus making the approach applicable to a broad range of offloading middleware. This aspect is a subject of future work as discussed in Chapters 4-6.

**Efficiency and Scalability:** Adaptive Offloading middleware target a range of adaptation scenarios ranging from small-scale collaborations adapting lightweight applications to larger-scale collaborations adapting computationally heavy systems. The proposed approach is applicable to both extremes because of the scalability of the approach with regards to application and collaboration sizes. In addition, the efficiency of the adaptation approach makes it ideal for a range of device capabilities.

However, while for reasons stated above the approach is expected to be generically applicable to existing computation offloading middleware, it is noted that the approach proposed in Chapter 3 and utilized in Chapter 6 depends on the existence of a multicast infrastructure that could limit the applicability of the proposed approach. Nevertheless, with the future rollout of IPV6 infrastructure, wherein multicast is enabled by default, it is expected that this limitation would be alleviated.
In terms of reproducibility of the work presented in this thesis, it is noted that the real-world applications considered are readily available as open source applications (SourceForge, 1999) and can be used to re-create the evaluation results presented. In addition, the runtime graphs generated and used during adaptation are available from the author upon request.

7.3 Future work and Limitations

This subsection identifies potential future work relating to the approaches proposed in this thesis, and other aspects of adaptive offloading closely related to the overall objective of this thesis. In doing so, some of the limitations of the proposed approaches discussed in earlier chapters, are reiterated so as to clearly identify future areas for improvement.

**Application Graph Contraction:** as discussed in Chapter 4, contracting the size of a hybrid granularity graph when required could curb the resource cost of the HGG approach. In the case of the Consolidated Adaptation Engine, CAE, this approach would be beneficial in scenarios where a device runs out of resources or the behaviour of an application evolves such that the efficacy gains of hybrid granularity are determined to be marginal. Similarly, the automatic determination and calibration of graph decomposition thresholds would allow for the optimization of the efficiency of HGG, based on various environmental and software execution contexts, and is hence a subject for future work (discussed in section 4.6).

**Adaptation Triggers:** Another interesting aspect of Adaptive Computation Offloading, which falls within the purview of the overall focus of this thesis, is the creation of efficacious adaptation triggers. Adaptation triggers are algorithms, which determine when adaptation decisions are required, based on external environmental factors. While a simple resource load based threshold scheme is widely adopted (i.e. trigger adaptation when resource utilization rises or falls below certain thresholds etc.) and was used in this thesis, the efficacy of such triggers could be improved. The efficacy of an adaptation trigger involves the balance between the required agility of adaptation (i.e. quick reaction to pertinent environmental changes) and the possibility of unnecessary adaptation, which would incur adaptation overheads that outweigh potential gains.

**Application Graph Models:** The representation of an application’s runtime for decision-making has notable impact on the efficiency and efficacy of adaptation as demonstrated through the approaches discussed in Chapters 4 and 5. Hence, future work could look at exploring different models of representing the execution of an
application’s runtime that better represents the execution behaviour of an application. One potential approach is a hyper graph model of representing an application in which, in addition to the ordinary vertices and edges of an application class (or hybrid) graph, hyper edges could be used to model the frequency of call chains in an application. This is expected to allow for the identification of the most frequent set of components involved in the invocation of specific business functionalities of an application, thus allowing for better optimization of adaptation objectives (e.g. performance of an application or energy utilization).

**Evaluations:** While this thesis considered a range of different evaluation scenarios for comparing the proposed approach to existing work, it is noted that more evaluations could be conducted to further verify and establish the results presented in this thesis. Specifically such evaluations could involve a larger corpus of applications with more diverse runtime behaviours, executing within a range of pervasive computing scenarios. In addition, such evaluations could include investigation of different use-cases of Adaptive Computation Offloading than the one focused on in this thesis (discussed in Chapter 1), such as improving reliability of mission critical systems or the availability of services in cloud computing environments.

**Fault Tolerance, Reliability and Security:** While this thesis focused on efficiency, scalability, and efficacy aspects of adaptive decision computation, the approaches proposed in this thesis could potentially be applied to address some of the other research challenges identified in Chapter 1, such as fault tolerance, reliability and security. Specifically, vertices in a component graph could be annotated with weights, which indicate the level of criticality of a component, its probabilistic likelihood of failure or the level of security it requires. The proposed adaptation engine would also need to be modified to collect and infer reliability and trust metrics within a collaboration environment. In conjunction, these software and environmental metrics could then be used to compute adaptation decisions that optimize the fault tolerance, reliability, and security attributes of an application, similar to the approach discussed in Chapter 3 and Chapter 4. In addition, a Hybrid Granularity Graph could be modified to group components based not only on their class types but also their security and fault tolerance requirements (similar to the approach in section 4.4.1) so as to optimize the efficiency of computing such decisions.
References


Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Gani, H., 2010. Transparent and adaptive application partitioning using mobile objects, School of Computer Science and Information Technology. RMIT University, Melbourne.


Mary Meeker, Scott Devitt, Wu., L., Internet Trends - Presentation from CM Summit.


Ou, S., Yang, K., Liotta, A., 2006. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems, Pervasive Computing and


Weisstein, E.W., Complete Graph.


Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L., 2010a. Accurate online power estimation and automatic battery behavior based power model