DYNAMIC SCHEDULING IN MULTICORE PROCESSORS

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Engineering and Physical Sciences

2011

By
Demian Rosas Ham
School of Computer Science
# Contents

Abstract 11

Declaration 12

Copyright 13

Acknowledgements 14

1 Introduction 15
   1.1 Motivation ..................................................... 15
      1.1.1 The End of the Uniprocessor Era ....................... 16
      1.1.2 The Multicore Era .................................. 19
      1.1.3 The Shift to the Many Core Era .................... 21
      1.1.4 Approaches for Exploiting Parallelism ............... 21
   1.2 Spreading Load with Active Messages ..................... 23
   1.3 Summary ....................................................... 25
   1.4 Research Aims ............................................. 26
   1.5 Contributions ............................................... 27
   1.6 Thesis Structure ........................................... 28

2 Background 29
   2.1 Introduction .................................................. 29
   2.2 Load Balancing .............................................. 29
      2.2.1 Work Sharing ........................................ 30
      2.2.2 Work Stealing ...................................... 30
   2.3 Active Messages ............................................ 32
   2.4 Lazy Task Creation ....................................... 34
      2.4.1 A simple example .................................... 36
## 2.5 Dynamic Scheduling

- **2.5.1 Software Scheduling** ........................................ 38
- **2.5.2 Hardware Scheduling** ........................................ 47

## 2.6 Summary ............................................................. 53

## 3 SLAM (Spreading Load with Active Messages)

### 3.1 Introduction to SLAM ........................................... 54

- **3.1.1 SLAM** .......................................................... 54
- **3.1.2 SLAM Task Stack** ............................................ 55
- **3.1.3 Load Balancing** .............................................. 56
- **3.1.4 Active Messages** ............................................. 56
- **3.1.5 SLAM Network** ............................................... 57
- **3.1.6 SLAM Communication Control** ............................. 57

### 3.2 How SLAM Works .................................................... 59

- **3.2.1 Evaluation of $Nfib(n)$** ...................................... 59

### 3.3 Programming with SLAM .......................................... 64

- **3.3.1 SLAM Programming Interface** .............................. 65

### 3.4 Analyzing SLAM Code ............................................. 67

### 3.5 Hiding SLAM Low Level Operations ............................. 68

- **3.5.1 A Simplified SLAM Programming Interface** ............... 68

### 3.6 Summary ............................................................. 70

## 4 SLAM Implementation ................................................. 71

### 4.1 Introduction ......................................................... 71

### 4.2 Task Stack .......................................................... 71

- **4.2.1 Task Stack Pointer** .......................................... 72
- **4.2.2 Task Stack Entry** ............................................ 72
- **4.2.3 Task Stack Operation** ........................................ 73
- **4.2.4 Task Stack Cache** ............................................ 74
- **4.2.5 Task Stack Cache Hardware** .................................. 75

### 4.3 Buffer of Stealable Tasks ........................................ 76

- **4.3.1 Buffer of Stealable Tasks Operation** ....................... 77
- **4.3.2 Buffer of Stealable Tasks Hardware** ....................... 78
- **4.3.3 Communication Control** ..................................... 79
- **4.3.4 SLAM Network** ................................................ 80
- **4.3.5 SLAM Hardware organization** ............................... 80
5.6.2 Parallel Evaluation ........................................ 109
5.6.3 Stealing and Running Time Distribution ............... 113
5.6.4 Steal Granularity Information .......................... 117

5.7 Results of CMPs with Shared Level 2 Caches ............. 122
5.7.1 Serial Evaluation ......................................... 122
5.7.2 Parallel Evaluation ...................................... 123
5.7.3 Stealing and Running Time Distribution ................ 127
5.7.4 Steal Granularity Information .......................... 128

5.8 A Deeper Look into SLAM ................................. 132
5.8.1 SLAM Relative Speedup ................................ 132
5.8.2 Execution Time of Private VS Shared L2 Caches ...... 136
5.8.3 Task Granularity of SLAM Applications ............. 138
5.8.4 Level 1 Stack Cache Read and Write Ratios .......... 138

5.9 Summary ....................................................... 140

6 Conclusions and Future Work ................................. 143
6.1 Conclusions ................................................ 143
6.2 Future Work ............................................... 145

Bibliography .................................................... 147
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>ISA extensions added by SLAM. The <strong>PUSH</strong>REM and <strong>OWRT</strong>REM operations only take into account the cycles spend by the CC to form the appropriate information packet. The time that the mentioned packet travels through the network is not included.</td>
<td>82</td>
</tr>
<tr>
<td>4.2</td>
<td>Messages involved in SLAM’s Dynamic Load Balancing.</td>
<td>90</td>
</tr>
<tr>
<td>4.3</td>
<td>Timing assumed for Communication Control operations.</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>Information provided by each simulation tool to the overall system simulation.</td>
<td>99</td>
</tr>
<tr>
<td>5.2</td>
<td>Inputs used in the system evaluation.</td>
<td>102</td>
</tr>
<tr>
<td>5.3</td>
<td>Configuration of the baseline simulated system.</td>
<td>107</td>
</tr>
<tr>
<td>5.4</td>
<td>Configuration of SLAM simulated systems.</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>Configuration of the single core baseline system.</td>
<td>108</td>
</tr>
<tr>
<td>5.6</td>
<td>Steals attempted and made when running SLAM applications using CMPs with private Level 2 caches.</td>
<td>118</td>
</tr>
<tr>
<td>5.7</td>
<td>Steals attempted and made when running Cilk applications using CMPs with private Level 2 caches.</td>
<td>118</td>
</tr>
<tr>
<td>5.8</td>
<td>Average, Maximum and Minimum steal granularities running SLAM applications using CMPs with private Level 2 caches.</td>
<td>119</td>
</tr>
<tr>
<td>5.9</td>
<td>Average, Maximum and Minimum steal granularities running Cilk applications using CMPs with private Level 2 caches.</td>
<td>119</td>
</tr>
<tr>
<td>5.10</td>
<td>Steals attempted and made when running SLAM applications using CMPs with shared Level 2 caches.</td>
<td>133</td>
</tr>
<tr>
<td>5.11</td>
<td>Steals attempted and made when running Cilk applications using CMPs with shared Level 2 caches.</td>
<td>133</td>
</tr>
<tr>
<td>5.12</td>
<td>Average, Maximum and Minimum steal granularities running SLAM applications using CMPs with shared Level 2 caches.</td>
<td>134</td>
</tr>
</tbody>
</table>
5.13 Average, Maximum and Minimum steal granularities running Cilk applications using CMPs with shared Level 2 caches. . . . . . . . 134
5.14 Task granularities and parallel phase lengths running SLAM applications using CMPs with private Level 2 caches. . . . . . . . . 139
5.15 Task granularities and parallel phase lengths running SLAM applications using CMPs with shared Level 2 caches. . . . . . . . . . 139
5.16 Read (R) and Write (W) Level 1 cache hit ratios when running SLAM applications using CMPs with private Level 2 caches. . . . 141
5.17 Read (R) and Write (W) Level 1 cache hit ratios when running SLAM applications using CMPs with shared Level 2 caches. . . . 141
List of Figures

1.1 Processor performance improvement from 1978 to 2004. . . . . . . 16
2.1 Work stealing scheme using a centralized queue of ready-to-execute
tasks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.2 Work stealing scheme using distributed queues of ready-to-execute
tasks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.3 Work stealing scheme implemented using distributed queues of ready-to-execute tasks and active messages to achieve task stealing
communication operations. . . . . . . . . . . . . . . . . . . . . . . 35
2.4 Binary tree used in the Lazy Task Creation example. . . . . . . . . 37
2.5 Pseudocode of a function that computes the sum of the leaves of
a binary tree recursively. . . . . . . . . . . . . . . . . . . . . . . . 37
2.6 Some stages of the parallel computation of the sum of the leaves
of a binary tree executed in a dual core machine. . . . . . . . . . 38
2.7 Cilk code example of the $nfib$ function. . . . . . . . . . . . . . . 41
3.1 Structure of the Task Stack. . . . . . . . . . . . . . . . . . . . . . 56
3.2 Code of the $Nfib(n)$ function. . . . . . . . . . . . . . . . . . . . . 60
3.3 $Nfib(5)$ computed in a single core machine using SLAM. . . . . . 61
3.4 $Nfib(5)$ computed in parallel in a dual core machine using SLAM. . . 62
3.5 Code of the $Nfib(n)$ function using SLAM functions. . . . . . . . 66
3.6 $Nfib(n)$ simplified code assuming SLAM low level operations auto-
mated. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.7 Comparison between $Nfib(n)$ simplified code and $Nfib(n)$ code that
includes SLAM low level operations. . . . . . . . . . . . . . . . . 69
4.1 Hardware structure of the Level 1 Task Stack Cache. . . . . . . . 76
4.2 Organization of the Level 1 Task Stack Cache. . . . . . . . . . . 77
4.3 Hardware structure of the Buffer of Stealable Tasks. . . . . . . . 79
4.4 SLAM Hardware organization. .................................................. 81
4.5 Actions performed by the CC when initiating the steal of a task. 83
4.6 Actions performed by the CC when serving a request for a task coming from a thief CPU. ............................................... 85
4.7 Actions performed by the CC when getting the answer corresponding to a request made to a victim CPU. ................................. 87
4.8 Actions performed by the CC when notifying a victim CPU about the completion of a stolen task. .......................................... 88
4.9 Actions performed by the CC when getting the notification about the completion of a stolen task from a thief CPU. ................. 89
4.10 $N_{fb}(5)$ computed in parallel in a dual core machine using the proposed SLAM implementation. ........................................... 94
4.11 Example CMP with SLAM hardware support. .......................... 96

5.1 Execution time of single thread execution of SLAM and Cilk versus C. ................................................................. 109
5.2 Speedup of SLAM and Cilk versus C for the divide-and-conquer parallel benchmarks tested using private Level 2 caches. ...... 111
5.3 Speedup of SLAM versus C for the loop-level parallel benchmarks tested using private Level 2 caches. ................................. 112
5.4 Stealing and running time distribution for the configurations with private Level 2 caches. $C$ and $S$ below the bars correspond to Cilk and SLAM respectively. .................................................. 115
5.5 Average steal granularity distribution with private Level 2 caches. 120
5.6 Execution time of single thread execution of SLAM and Cilk versus C. ........................................................................... 123
5.7 Speedup of SLAM and Cilk versus C for the divide-and-conquer parallel benchmarks tested using shared Level 2 caches. ....... 125
5.8 Speedup of SLAM versus C for the loop-level parallel benchmarks tested using shared Level 2 caches. ................................. 126
5.9 Stealing and running time distribution for the configurations with shared Level 2 caches. $C$ and $S$ below the bars correspond to Cilk and SLAM respectively. .................................................. 129
5.10 Average steal granularity distribution with shared Level 2 caches. 130
5.11 SLAM relative speedup when running in CMPs with private Level 2 caches. ................................................................. 135
5.12 SLAM relative speedup when running in CMPs with shared Level 2 caches. ................................................................. 135
5.13 SLAM execution time breakdowns for the divide-and-conquer parallel applications .................................................. 137
5.14 SLAM execution time breakdowns for the loop-level parallel applications ................................................................. 138
Abstract

The advent of multi-core processors, particularly with projections that numbers of cores will continue to increase, has focused attention on parallel programming. It is widely recognized that current programming techniques, including those that are used for scientific parallel programming, will not allow the easy formulation of general purpose applications. An area which is receiving interest is the use of programming styles which do not have side-effects. Previous work on parallel functional programming demonstrated the potential of this to permit the easy exploitation of parallelism.

This thesis investigates a dynamic load balancing system for shared memory Chip Multiprocessors. This system is based on a parallel computing model called SLAM (Spreading Load with Active Messages), which makes use of functional language evaluation techniques. A novel hardware/software mechanism for exploiting fine grain parallelism is presented. This mechanism comprises a runtime system which performs dynamic scheduling and synchronization automatically when executing parallel applications. Additionally the interface for using this mechanism is provided in the form of an API. The proposed system is evaluated using cycle-level models and multithreaded applications running in a full system simulation environment.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://www.campus.manchester.ac.uk/medialibrary/policies/intellectual-property.pdf), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to thank the National Council of Science and Technology of Mexico for sponsoring my PhD studies. I would also like to thank my supervisor Professor Ian Watson for the help, support, encouragement and feedback he has given me during the last four years.

I would like to thank Isuru Herath and Paraskevas Yiapanis with whom I have shared my time during my studies. Their friendship and support made my time in Manchester very enjoyable. I thank all the members of the APT group with whom I had interesting discussions about research and non-research topics.

I thank my parents and my brother for their support and encouragement.

Special thanks to Claudia for her unconditional encouragement, patience and support.
Chapter 1

Introduction

An efficient dynamic load balancing system for shared memory chip multiprocessors is presented in this thesis. This system allows the exploitation of fine grain parallelism on multithreaded applications. Dedicated hardware support is provided in order to improve performance, as well as to achieve load balancing at runtime. The presented system is modelled using a full system simulation platform and is used to evaluate concepts regarding performance, scalability, workload distribution and load balancing.

1.1 Motivation

For nearly 30 years computer industry created each time faster computers due to the advances in Integrated Circuits technology. As established by Moore’s law in 1965, the number of components integrated on a chip doubled every 12 months until 1975, when Moore adjusted his law to establish that the following years the number of components would double every two years [61, 4]. The accuracy of this prediction allowed the computing industry to produce faster computers relying on building higher performance uniprocessors. To do this, each time better chip fabrication methods available were used, as well as more sophisticated computing design techniques which emphasized architectural innovation coupled with an efficient use of technology improvements [35].
1.1.1 The End of the Uniprocessor Era

Figure 1.1 shows microprocessor performance improvement since 1978 up to 2004. There are three well defined regions. The first shows an improvement of 25% per year from 1978 to 1986. The second region shows an improvement of 52% per year from 1986 to 2002, which has been the most prolific age for computing so far. The third region shows a more modest improvement of around 20% per year from 2002 to 2004, the rate that prevails until today [36].

This drastic reduction in performance improvement is explained by the inherent challenges faced when designing modern microprocessors. The main challenges faced when designing a microprocessor nowadays are wire delay, memory gap, limited instruction level parallelism, power consumption and design complexity.

Wire Delay

With the advances in semiconductor technology, transistors are continually getting smaller, allowing a higher number of them per silicon die, having more transistors available for implementing more functionalities per chip (deeper pipelines, wider register windows, better branch predictors [58], more hardware contexts [53], etc.)
CHAPTER 1. INTRODUCTION

44, 63], etc.) [35]. However the wires used to connect those transistors have necessarily to shrink as well. The problem comes with the fact that the wire resistance increases with the reduction of the size of the wires. This effect means that only a limited section of an entire chip is reached in a single clock cycle, due to the inherent delay of the on-chip signals caused by the bigger electrical resistance [39].

Memory Gap

For nearly three decades microprocessor performance was driven mainly by increasing the clock frequency. However, memory performance did not improve at the same rate. This fact limits microprocessor performance due to the existing difference between processor and memory speeds. In modern microprocessors a single off-chip memory access takes several hundreds of cycles. This performance difference reduces the benefit of having a high clock frequency, since a CPU that gets data from memory spends a significant amount of cycles idle waiting for the off-chip memory response [35].

In order to tackle this problem cache memories located in the same chip as the CPU, which are smaller and much faster than their off-chip counterparts, are used to exploit data and temporal locality when executing an application. This serves to hide off-chip memory latency, since the most frequently accessed data is held in the cache memories.

Limited Instruction Level Parallelism

Modern microprocessors include hardware for exploiting Instruction Level Parallelism (ILP). ILP comes from the fact of executing multiple instructions independent from each other simultaneously. Super scalar processors exploit ILP issuing multiple instructions per cycle and, if the dependencies among these instructions are solved dynamically in the Decode stage, they belong to the out-of-order (OOO) processors class. VLIW (Very Large Instruction Word) processors execute multiple instructions simultaneously, but the compiler is responsible for identifying the sets of instructions that can be executed in parallel.

Instructions in the same thread normally exhibit a high level of dependence. This fact makes necessary to analyze in advance the instruction stream in order to find independent instructions to execute in parallel. In the case of the superscalar architectures, the above fact involves a significant increase in hardware resources with corresponding added complexity. In the case of VLIW processors,
the compiler frequently cannot form VLIW instructions that make use of all the functional units of the CPU, leading to subutilization.

Studies in the past have shown that the number of instructions that can be executed simultaneously is on average 5, with a maximum of around 7 [77, 16, 78].

Power Consumption

In modern microprocessors power consumption has become a first order factor when designing a new architecture.

The power consumed by a microprocessor has two components: a static one called the leakage power, and a dynamic one called switching power. These components are shown in Equation 1.1. The reduction in size of transistors, leads to a reduction of their capacitance (C), which in turn leads to shorter switching times and in consequence, higher performance due to the higher operation clock frequency. However, as can be seen from Equation 1.1 $Power_{total}$ is directly proportional to the clock frequency ($f_{clock}$) which is included in the switching power ($Power_{dynamic}$) of the chip.

Also with the reduction in transistor size, the voltage (V) has to be lowered in order to meet thermal requirements for the design. This reduction in the voltage (V) forces a reduction in the threshold voltage at which transistors switch their state in order to keep the chip working properly. This lower threshold voltage makes the whole chip to work closer to ground levels, which increases the static leakage current ($I_{leakage}$), which in turn increases static power consumption.

$$Power_{total} = Power_{static} + Power_{dynamic} = V I_{leakage} + CV^2 f_{clock} \quad (1.1)$$

Design Complexity

The design complexity inherent in modern microprocessors comes from the fact that these integrate more functionality aimed to get better performance. The fact of including deeper pipelines, better branch predictors [35], simultaneous multithreading [53], etc. makes the verification of the systems a very difficult task [5]. However this high level of complexity has not delivered a proportional benefit in terms of performance, as can be seen in Figure 1.1, considering that the tendency shown up to 2004 is still prevalent at the present [36].
1.1.2 The Multicore Era

The task of building complex single core processors, in the search for higher performance, became infeasible due to the small performance gain, consequences of the difficulties regarding design complexity, power consumption and limited instruction level parallelism. In the light of this situation, a new architecture was proposed, one that integrated multiple cores in a single chip. This architecture was given the name of Chip Multiprocessor (CMP) [65].

A CMP comprises two or more cores which are independent from each other and can work either, cooperatively for solving a single problem, or separately on different problems. If all the cores in the CMP are identical, then it is called an homogeneous CMP [44, 63, 41, 10]. If a CMP contains different types of cores, then it is called heterogeneous [47, 70, 40].

Among the advantages of CMPs over single core superscalar processors are that the engineering effort is smaller for the first ones. This is due to the fact that the CMP is built using cores that are already designed and validated. These cores only have to be placed in the chip and interconnected. Also, since CMPs are made of simpler cores that operate at lower frequencies than their single core counterparts, they are more power efficient. In terms of design scalability, a CMP can be expanded basically adding more cores to a single chip, with the corresponding interconnection logic [41, 10], keeping the overall hardware simple.

Programming Multicore Processors

For exploiting parallelism when executing a single application using CMPs, multiple threads must be used so that they can be mapped into multiple cores for achieving parallel processing. This fact makes necessary for applications to be multithreaded in order to take advantage of the resources offered by CMPs [56]. The problem with this is that in order to develop parallel applications for CMPs, programmers must be aware of the existence of multiple cores. This implies that programmers have to divide the corresponding applications in sections that can be mapped to multiple cores to be executed concurrently.

In a multithreaded application, a thread is a sequential process that shares the same memory space with the other threads. Threads are a model of concurrency supported by computers, programming languages and operating systems [50]. Some applications known as embarrassingly parallel can make a very efficient use of threads (scientific applications, web servers, etc.). These kind of applications
comprise a set of independent processes that can be executed simultaneously. Due to the independence among the processes executed, programming embarrassingly parallel applications is relatively easy, since the synchronization involved is minimal. For the reasons above this kind of application can take advantage of the existence of multiple cores, showing faster executions as more cores are available on the system in the presence of enough parallelism.

The situation regarding mainstream applications is different. These applications are fundamentally single threaded meaning that in modern multicore processors, they often run slower. This is because modern CMPs are built with simpler cores that run at slower frequencies. In order to exploit the processing power provided by a CMP, applications must be highly concurrent, meaning that they must use multiple threads that can be mapped to different cores and executed simultaneously.

The fact that threads which belong to a single application share the same memory space, means that any modification to the memory made by one of the participant threads becomes visible to all the other ones. In a situation in which two threads access the same data structure or object in memory and one of them updates the state of it, a race condition may arise if no synchronization mechanism is provided. When writing parallel software, race conditions are unacceptable since these may lead to inconsistent states during the execution of an application. Because of this, synchronization is necessary to guarantee execution correctness in parallel applications whose threads work concurrently on the same data [72].

The Problem with Threads

The problem with using threads for producing parallel applications is the non-determinism inherent to them. When using threads, the order in which these are scheduled for execution by the operating system is unpredictable. There are mechanisms to deal with this non-determinism such as semaphores, mutexes and monitors. The purpose of these mechanisms is to provide the programmer with tools for implementing synchronization among multiple threads in a single application. However, it is widely agreed that these mechanisms lead to code which is difficult to understand when working with non-trivial parallel applications. Additionally programming multithreaded applications using locks is complex and error prone, making multithreaded programs likely to contain concurrency bugs [50].
CHAPTER 1. INTRODUCTION

Alternative Tools

In the light of the above situation numerous tools for writing parallel applications have been created. In the field of high performance computing, libraries like MPI [1] and OpenMP [2] have been used for a long time to write parallel programs. These tools are appropriate for writing applications aimed to exploit parallelism in very regular applications that have the parallel nested For All form. However for not so regular applications other tools are more appropriate. In order to write applications in which parallelism is irregular, tools such as Cilk [29, 14], Cilk++ [52], Intel TBB [46], OpenMP Tasks [8, 23], Wool [26, 27], Stack Threads [73], the Java Fork/Join Framework [49], SLAW [30], the Microsoft Parallel Library [51], among others have been developed for generating parallel applications in which the exploitation of parallelism does not follow a regular pattern.

These tools are aimed to provide mechanisms for exploiting parallelism efficiently on shared memory CMPs. They provide functionality for exploiting loop and task level parallelism, as well as dynamic load balancing capabilities that provide a higher level abstraction to the programmer than that given by the threads programming model. These tools are designed to work with much finer grain parallelism than that present on matrix and array scientific applications, or server applications such as web or database servers.

1.1.3 The Shift to the Many Core Era

Now that multicore processors are the mainstream, researchers are focusing on chips with dozens, hundreds of even thousands of cores [41, 10, 24], dynamic scheduling of fine grain parallelism is getting more attention. In order to achieve an efficient load balancing, systems capable of distributing the workload of parallel applications in an environment with dozens or hundreds of cores are going to be required.

1.1.4 Approaches for Exploiting Parallelism

As mentioned in the previous sections, in order to take advantage of the processing power of multi and many core systems, parallelism must be exploited. There are different types of parallelism and they are also exploited at different granularities.
CHAPTER 1. INTRODUCTION

Instruction Level Parallelism (ILP) is achieved when several instructions are executed simultaneously in the processor’s pipeline. In order to exploit ILP, the sequences of instructions need to be independent from each other and they also must make use of different functional units in the CPU simultaneously. Besides executing several instructions in parallel, the objective of ILP is also to keep the functional units of the CPU as busy as possible.

Data Level Parallelism (DLP) is achieved when the same operation is executed on multiple data sets simultaneously. Streaming applications are a good example of the exploitation of DLP. For example, when applying an operation to a digital image, the same operation is performed on several pixels in parallel, these pixels are independent from each other. Also when performing operations on matrices, arrays and vectors, DLP can be exploited. When processing a matrix or an array using a loop and, a section of the matrix or array can be assigned to a different CPU to be processed in parallel with the other sections, this exploitation of parallelism is performed at loop-level.

Thread Level Parallelism (TLP) is the one existent in an application that runs multiple independent threads concurrently. Applications like web servers and data base servers exploit this kind of parallelism since these applications work with multiple requests that are independent from each other. These kind of applications are generally run on machines with a high workload. A particular case of TLP is task-level parallelism. It is found when applications are expressed as a set of independent tasks that form a graph of computation, in which the edges of the graph represent the data dependencies. These kind of applications exploit parallelism at task-level.

Task Level Parallelism

Nested task parallelism is a programming model that has received a lot of attention due to the arrival of multicore processors [29, 14, 52, 46, 8, 23, 26, 73, 49, 30, 51]. In this model, parallelism is expressed in the form of tasks that are allowed, but not necessarily are executed in parallel. This model of parallelism was identified as a requirement for success [30] when developing new programming languages [17, 3, 18] as part of the DARPA’s High Productivity Computing Systems project.

When using dynamic task parallelism, the tasks to be executed are created dynamically, a runtime system is then responsible for the scheduling of the created
tasks across the cores available in the system. In order to achieve good performance when using dynamic task parallelism, the scheduler must be efficient and scalable. Several software tools implementing this programming model have been developed [29, 14, 52, 46, 8, 23, 26, 27, 73, 49, 30, 51].

Work stealing [13] is a technique employed for achieving dynamic load balancing. Popular work stealing schedulers [29, 14, 52, 46, 26, 27, 73, 49, 30, 51] make use of a number of threads called workers. The number of the workers matches the number of cores available for the parallel computation. Each worker has assigned a local double-ended queue in which the tasks are stored when created and retrieved when executed. These operations are performed at the bottom of the local queue. When a worker runs out of tasks, or it is suspended at a synchronization point and there is no local task available for execution, it tries to steal a task from another worker and when the steal is successful, the task is taken from the top of the victim worker’s doubled-ended queue.

In shared memory CMPs with dozens of cores, dynamic scheduling implemented purely in software is unlikely to be capable of exploiting fine grain parallelism efficiently [48, 22]. This is because these kind of schedulers make use of the shared memory for performing load balancing related operations. The problem with this is that as the number of cores in the chip increases, the traffic of load balancing messages is also increased which in turn, increases the number of memory accesses with corresponding performance degradation.

Previous studies have suggested that the use of hardware acceleration mechanisms would be beneficial in order to improve scalability and performance in many core dynamic schedulers [48, 22]. Hardware assisted dynamic scheduling systems make use of the Network-on-Chip in order to send and receive messages related to load balancing operations. This use of the processor interconnect avoids memory accesses and this in turn is reflected in better performance and scalability.

1.2 Spreading Load with Active Messages

Much work has been done in the past on the use of functional programming [42] to exploit parallelism. Much of the complexity of parallelism comes from the handling of shared state and its absence from the functional style enables parallelism to be exploited with relative ease. The drawback of the approach is that computations with mutable state are difficult and/or inefficient to express.
Nevertheless, with the shift to multicore and many core processors, interest in this field has been revived. One major development is the availability of systems which use conventional languages but provide a framework which can efficiently exploit parallelism if the programs are expressed in a largely functional style. One of the best known of these is Cilk [29, 14].

The functional style permits the expression of parallel programs which are often much more irregular and dynamic than many scientific ones and this is an advantage for more general applications. However, it does lead to both lower granularity parallelism and the need for careful runtime management of resources. Dynamic load balancing is an important part of the Cilk system and has been shown to produce efficient use of parallel CPUs [29, 14]. A significant amount of other research has been done on dynamic load balancing for multiprocessor machines [52, 46, 8, 23, 26, 27, 73, 49, 30, 51].

In order to produce an efficient evaluation mechanism for an early Object Oriented Functional Language called UFO [67], a dynamic load balancing system known as SLAM (Spreading Load with Active Messages) was developed [81]. This was, in turn, based on packet based graph reduction techniques for functional languages [80]. In the work done in this thesis SLAM has been adapted in order to make it work in the context of modern CMPs executing imperative programs.

The essence of packet based graph reduction is to represent a function applied to arguments as a memory based structure. A complete program is a graph composed of packets as nodes with the edges representing the data flow between them. Evaluation proceeds by taking individual packets and executing code to perform the function. The code can be compiled into a normal sequential instruction stream thus allowing implementation on conventional processors. Parallelism is achievable if the graph has multiple packet nodes which are ready for evaluation at any given point in time.

The fine grain nature of the individual functions makes even spreading of load, across processing resources, relatively easy. Unfortunately, the level of granularity has a negative effect of increasing significantly the scheduling costs if individual function evaluations are spread randomly across parallel resources. The aim of SLAM is to constrain the distribution of work in a way which potentially allows fine grain parallelism but groups closely connected areas of graph to minimize communication and scheduling costs. This is achieved by mapping the graph structure on to parallel stacks which exist in each processor. In most cases,
scheduling takes place in stack order and values are communicated within the stack. Parallelism is spread by processors stealing from other stacks if they are short of local work, in most cases this results in a newly growing stack on the stealing processor.

The process of scheduling execution, communicating values and work stealing requires a significant amount of housekeeping computation which is separate from the normal evaluation. Instead, SLAM embeds this control in the normal execution mechanism using techniques which are very similar to Active Messages [76]. The result of this is that the complete execution cycle is simply to extract an entry from the top of the stack and jump to the code which is pointed to by its function pointer field. By this means, the overheads of the control are minimized.

With billions of transistors available to build the future CMPs, it is valid to consider if hardware support is appropriate to support parallel evaluation mechanisms such as SLAM. Therefore in this thesis, simple hardware extensions have been studied and evaluated in order to increase the efficiency of SLAM. These are mainly concerned with user level inter core messages and stack caches for the main evaluation control structure. Other approaches using custom hardware for exploiting fine grain parallelism have been explored in Carbon [48] and Asynchronous Direct Messages [22].

1.3 Summary

This chapter has presented a review of the factors that led to the adoption of Chip Multiprocessors as the vehicle to build more powerful computers. It mentioned the challenges that made it infeasible to continue building uniprocessor architectures. Issues concerning the wire delay, memory gap, power consumption, limited instruction level parallelism and design complexity were described as the reasons that motivated the shift towards Chip Multiprocessor architectures.

Chip Multiprocessors were also described mentioning the set of advantages they brought to the computer architecture arena. These advantages are a smaller engineering effort due to the use and replication of already validated cores with relatively simple interconnection logic, lower frequency operation coming from the fact of using simpler cores, getting in turn more power efficient architectures. Additionally, CMPs provide design scalability since expansion can be achieved by adding more cores into a chip, with the corresponding interconnection logic,
CHAPTER 1. INTRODUCTION

keeping the overall hardware simple.

This chapter also pointed out the need of having multithreaded applications in order to exploit the computing power offered by CMPs. The threads programming model was described as a non-convenient model for writing parallel applications due to its inherent non-determinism. Also a set of parallel programming tools that provide higher level mechanisms for producing parallel applications were mentioned. These tools provide a higher level abstraction for writing concurrent applications and in turn are more appropriate for exploiting concurrency than the threads programming model.

The shift to having many core architectures was also discussed. As more cores are added to the chips, more efficient and scalable load balancing systems are required. The presence of CMPs with higher numbers of cores has pushed the study of dynamic task-based parallel programming models running on top of load balancing systems capable of distributing the workload at runtime. Systems that avoid the use of the shared memory for performing load balancing activities have been proposed as alternatives for achieving the required better efficiency and scalability.

Finally this chapter mentioned the SLAM (Spreading Load with Active Messages) system as an alternative for generating parallel applications, in which the exploitation of fine grain parallelism is possible. SLAM is based on packet graph reduction techniques, used in the past for the parallel evaluation of functional languages.

The next chapter provides the necessary background information for understanding the SLAM system. It explains the concepts on which SLAM relies such as Dynamic Load Balancing, Work Stealing [13], Lazy Task Creation [60] and Active Messages [76]. A review of the related work done in both, academia and industry is also presented.

1.4 Research Aims

The research presented in this thesis is focused on the implementation and evaluation of SLAM in the context of modern shared memory Chip Multiprocessors. Relatively simple hardware support is added in order to make SLAM’s operation more efficient. These hardware additions comprise a task stack cache system and a dedicated communication mechanism that deals with the load balancing
messages at execution time. This evaluation is performed using simulated cache coherent 2-Dimension mesh tiled CMP architectures using two different configurations. In the first one each tile has a private block of Level-2 cache. In the second configuration each tile contains a Level-2 cache block that is shared with all the other cores of the chip.

1.5 Contributions

The contributions made in the work presented in this thesis are summarized as follows:

- A dynamic load balancing system based on SLAM adapted to work in tiled shared memory Chip Multiprocessors.

- Appropriate hardware support for increasing the efficiency of the load balancing system, making it capable of exploiting fine grain parallelism when required, but being also able to work with coarse grain applications.

- A fully cache coherent evaluation of the proposed SLAM implementation using simulated Tiled Shared Memory Chip Multiprocessors ranging from 4 to 64 cores.

- A full system simulation platform for SLAM. This platform includes operating system support, directory-based cache coherent memory system, point-to-point mesh processor interconnection network and SLAM hardware acceleration support.

- A SLAM emulator. This is an emulator capable of running multithreaded SLAM applications on real multiprocessor machines (without SLAM hardware support).
1.6 Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2 provides background information comprising the concepts on which SLAM relies. Concepts such as Dynamic Load Balancing, Work Stealing, Lazy Task Creation and Active Messages are described. Also a review of the related work is presented.

Chapter 3 describes the SLAM system at a conceptual level. It describes the way in which work stealing, lazy task creation and active messages are combined in order to achieve dynamic load balancing. It also explains the way in which the task stack data structure is used in order to achieve synchronization when executing a parallel computation.

Chapter 4 describes the actual implementation of SLAM developed in this thesis. It explains the way in which the custom hardware additions are used for exploiting parallelism in shared memory Chip Multiprocessors using SLAM.

Chapter 5 presents the evaluation of the system. It introduces the tools used for simulating the multiprocessor system environment, as well as the custom hardware modeling. It also presents a performance comparison of a system with SLAM implemented versus a base line system running Cilk [29, 14]. Finally it provides a discussion about the results obtained from the experiments performed during the evaluation of SLAM.

Chapter 6 concludes the thesis based on the results shown in the evaluation chapter. It also mentions future research directions to follow in order to make a more detailed exploration of SLAM as an alternative for writing parallel applications for current and future multicore systems.
Chapter 2

Background

2.1 Introduction

The purpose of this chapter is to provide a comprehensive explanation of the concepts related to the work developed in this thesis. The description of concepts like 
Load Balancing, Work Stealing, Active Messages and Lazy Task Creation is presented. These concepts are the base on top of which SLAM [81], the parallel computing model studied in this thesis is built.

This chapter also presents a review of the work related to this thesis. This review covers computing systems aimed at exploiting multiple cores through the use of dynamic scheduling of computation of user level threads.

A description of systems such as Cilk [29, 14], Threading Building Blocks [46], OpenMP Task [8] among others is presented in the section related to software scheduling systems.

Also a description of dynamic scheduling systems which make use of dedicated hardware for accelerating scheduling, as well as for exploiting fine grain parallelism is provided. This description covers systems such as Carbon [48] and ADM [37] among others.

2.2 Load Balancing

Load balancing as the name suggests, is the action of balancing the workload among different workers such that each performs a similar amount of work. Similarly, in the context of multicore computing systems, load balancing refers to the distribution of a given computation among the available processing nodes, such
that each node performs a similar amount of work to the others. Load balancing can be performed in parallel computing systems either statically or dynamically.

In order to do static load balancing, the computation to be executed by a multicore system must be distributed from the beginning. In this way each processing node is given a portion of the computation by a CPU that plays the role of the master. Once a worker finishes its portion of work, it notifies it to the master. When this gets the notifications of all the processing nodes involved in the computation, it can proceed with the next stage of it. Static load balancing is appropriate when the computation to be performed is well known in advance, so that a static partitioning of it can be made. An example of this are the applications that have the parallel nested For All form of many array algorithms.

There are cases in which the computation to be performed is not known in advance. In these cases applications have an irregular behavior that changes constantly at runtime. For this kind of situation static partitioning is not appropriate, since it leads to load imbalance among the processing nodes. Here is where dynamic load balancing comes into the scene. As its name suggests, this kind of load balancing aims to balance the workload of a parallel computation dynamically. The load balancing in this case is performed at runtime, in which the workload is distributed following specified criteria. Two examples of dynamic load balancing techniques are work sharing and work stealing.

### 2.2.1 Work Sharing

In work sharing, every time a processor generates new tasks for execution, the scheduler tries to distribute some of them to other processors. This in the hope that underutilized processors get some work to do, keeping in this way all the processing nodes doing useful work.

### 2.2.2 Work Stealing

In work stealing [13], a processor that runs out of executable tasks, tries to steal some work from another processor chosen by a certain criterion. If the steal is successful, then the processor that stole the work proceeds to execute it. Otherwise the stealing attempts continue until a successful steal is achieved.
Work Stealing with a Centralized Queue

The work stealing scheme can be implemented in different ways. One of these is using a centralized queue of ready to execute tasks. In this implementation, when a processor generates a task but cannot execute it immediately, it places the generated task in the centralized queue. In this way if there is an idle processor trying to steal some work, it would be able to steal the recently created task. The problem with this implementation is that the centralized queue can become a bottleneck as the number of processors in the system increases. This is due to the contention generated when several processors try to steal from the queue simultaneously. Figure 2.1 shows the work stealing scheme implementation using a centralized queue of ready-to-execute tasks.

Figure 2.1: Work stealing scheme using a centralized queue of ready-to-execute tasks. The arrows in the figure indicate the flow that ready-to-execute tasks follow in the stealing process.
Work Stealing with Distributed Queues

In order to solve the contention problem generated by having a single centralized queue of ready-to-execute tasks, the mentioned queue can be distributed among all the processors in the system. In this scheme each processor has attached a queue of ready-to-execute tasks. The new generated tasks are stored in this queue by each processor. When a processor runs out of executable tasks in its local queue, then it selects a victim using a certain criterion and tries to steal a task from it. In this way the processor that tries to steal becomes a thief, while the other processor involved is the victim. If the steal fails because there is no ready-to-execute task in victim’s queue, then the stealing process is repeated trying with a different victim until it succeeds. Figure 2.2 shows the work stealing scheme implementation using distributed queues of ready-to-execute tasks.

2.3 Active Messages

The Active messages approach [76] is an asynchronous communication mechanism designed to work in multiprocessor machines. This mechanism is intended to perform communication operations keeping the associated overhead at very low levels. This is possible due to the overlapping of computation and communication operations that results from the use of this mechanism.

In this communication mechanism each message contains at its head the address of a user level handler, which is executed on arrival by the receiving processing node. The associated handler arguments are contained in the body of the message.

What the handler does is to extract the message from the network, for later making it part of the ongoing computation of the receiving processing node. This handler must be executed quickly and to completion.

Work Stealing using Active Messages

Active messages is a convenient mechanism for implementing a work-stealing scheme with distributed queues in a multiprocessor system. In order to do this, the messages must contain steal requests from the side of the thieves, as well as steal request answers from the side of the victims. In order to make this scheme efficient enough for making it work in a multiprocessor system, there must exist
certain conditions:

- The message handlers must be very short routines, in the order of dozens of cpu-cycles, capable of executing the required work-stealing functionality.
- The communication network in which the messages involved in work-stealing
travel, must be a low-latency network.

- The amounts of work stolen by the thieves in successful steals have ideally to be big enough, so that the communication overhead is kept at low levels.
- The refusals produced by the victims due to unsuccessful steals must get to the thieves quickly, so that they can keep trying to get work from different victims.

Figure 2.3 illustrates a work stealing scheme with distributed queues that makes use of active messages to perform task stealing communication operations. In (a) Processor 1 (P1) has a local queue holding many ready-to-execute tasks. On the other hand, Processor 2 (P2), which has no tasks in its local queue, sends an active message to P1 requesting some work to do. In (b) P1 extracts the message from the network, getting in this way P2’s request. In (c) P1 executes the appropriate message handler, sending the task n to P2. Finally in (d) P2 extracts the message from the network, and executing the appropriate message handler, places task n in its local queue. In this situation P1 and P2 can process their respective ready-to-execute tasks in parallel.

2.4 Lazy Task Creation

Lazy Task Creation [60] is a technique used in order to increase the granularity of parallel programs. If the average runtime granularity of a parallel program is large in comparison with the overhead of task creation and scheduling, then the program execution is efficient.

When using lazy task creation, every potentially parallel task is created to be executed inline, but saving enough information so that these tasks can be unlined if, during the execution, processing resources become available. Applying this technique most of the tasks created in a program are executed inline, but they are ready to be stolen cleanly by another processor in case it gets idle. In this way, when executing a parallel computation the number of stolen tasks is small.

In a fine-grain program the execution tree may have an abundance of potential fork points. Using lazy task creation only a small subset of these potential forks are handled as actual forks. In this way runtime granularity is maximized, whilst at the same time parallelism is preserved so that good load balancing
Figure 2.3: Work stealing scheme implemented using distributed queues of ready-to-execute tasks and active messages to achieve task stealing communication operations. The arrows going from the processors to the queues and vice versa indicate the flow that ready-to-execute tasks follow in the stealing process.
can be achieved. Since only a small number of tasks are migrated from their
owner processors, the communication overhead of the overall system is reduced,
contributing to a better performance.

2.4.1 A simple example

In order to illustrate the way in which lazy task creation is applied when per-
forming a parallel computation, consider the recursive algorithm used to sum the
leaves of a binary tree executed in a dual core machine. Figure 2.5 shows the
pseudocode of a function that performs the above computation. The binary tree
shown in Figure 2.4, in which all the nodes are named, is used to illustrate this
example.

The notation used in this example is described below:

- \text{st}(\text{node}) is a call to the function \text{st} passing \text{node} as its argument.
- \text{node} refers to the corresponding node in the binary tree, it can be a single
  leaf or a subtree..

In this way, a call \text{st}(R) means a call to \text{st} passing as argument the subtree,
with root in the node \text{R} of the tree.

Figure 2.6 shows some stages of the parallel computation of the above problem.
In (a) P1 has executed some recursive calls to \text{st}. These calls are stored in its
local stack, including those tasks that could be executed in parallel. In (b) P2
has stolen the task corresponding to the call \text{st}(R) and performed the associated
calls. Notice that this task was created lazily, meaning that it was created only
because there was a processing resource available. Otherwise it would have been
preserved in P1’s pool of tasks and, at the proper time, executed by it serially.

If there were more processors available, more parallel tasks would be lazily
created for them, but only until all the processors had some work to do. There-
after, more tasks would be created only if some of those processors got idle. In
this way, since only a small subset of all the potentially parallel tasks are actu-
ally made parallel, the granularity of the parallel computation is increased. In
consequence, the involved processors spend longer time doing useful computation
and, in the presence of enough parallelism, only spend a short time performing
communication operations.
Notice the *WAITING* statement placed in P1’s pool of tasks. This indicates that a result is expected and that the computation can proceed only after this result is ready. In this way the correctness of the computation is preserved.

It is also worth noticing that since the task stolen from P1 was at the bottom of the pool, it was the biggest task available for migration. This is a feature that is present and can be exploited in divide and conquer problems. Considering that there are only two processors in this example, a possible (and ideal) division of the work done is illustrated by the way the nodes in Figure 2.4 are coloured. In this case, white nodes are the ones computed by P1 and the black ones are those computed by P2.

![Figure 2.4: Binary tree used in the Lazy Task Creation example.](image)

```plaintext
st(tree){
    if tree is a leaf
        return leaf_value
    else
        return(st(left_tree) + st(right_tree))
}
```

Figure 2.5: Pseudocode of a function that computes the sum of the leaves of a binary tree recursively.
Figure 2.6: Some stages of the parallel computation of the sum of the leaves of a binary tree executed in a dual core machine.

2.5 Dynamic Scheduling

All the concepts described previously are techniques used in order to achieve an efficient scheduling of parallel computations in a multicore computer. In particular, these techniques are applied for implementing dynamic scheduling systems. In a multicore computer, the scheduling system is responsible of achieving an efficient load balancing with the aim of getting the best possible performance and, as the name suggests, it operates at runtime.

The runtime scheduling of the workload in a parallel computation can be implemented entirely in software, entirely in hardware, or using a combination of both. The remaining of this section provides a description of some of the most representative software, hardware and hardware-software dynamic scheduling systems that have been proposed in order to achieve load balancing when performing parallel computations in multicore computers.

2.5.1 Software Scheduling

Schedulers implemented entirely in software maintain queues of ready-to-execute tasks in memory. Each of these queues is attached to one processor which enqueues and dequeues tasks locally. Communication among threads to do load balancing is made through the shared memory.

The advantage of these kind of schedulers is that they require no hardware
Modification for their implementation and, also their inherent flexibility allows
the use of different scheduling policies. On the other hand, these schedulers
offer a limited scalability when the computation involves very fine grain tasks.
This is because the scheduling overhead is comparable to the useful computation
performed in this kind of applications.

Cilk

Cilk [29, 14] is a runtime system for multithreaded parallel programming based
on the C language. In Cilk, a computation can be thought as a direct acyclic
graph (dag) that unfolds at runtime. A Cilk program is conformed by a group of
Cilk procedures. These procedures are broken into a sequences of threads, which
in turn form the vertices of the dag.

A thread in this context is a nonblocking C function (it can run to comple-
tion once being invoked). When running, a thread invoked in a Cilk procedure
can spawn a child thread starting in this way a new Cilk procedure. The spawn
keyword is used to indicate that a Cilk procedure can continue its execution in
parallel with its child. In this context, a spawned procedure in Cilk is similar to
a C function, with the difference that the latter cannot be executed in parallel
with other functions. The first Cilk version called Cilk-1 [14] used a random-
ized work stealing scheduler [13] and parallelism had to be exposed using explicit
continuation passing. Version 5 of Cilk, called Cilk-5 [29], uses a similar sched-
uler, but the language was simplified. It implements call/return semantics for
parallelism. It also includes an inlet mechanism (explained later in this section)
for nondeterministic control. Cilk runs efficiently in shared memory symmetric
multiprocessors.

The Cilk language is a parallel extension of C. The Cilk control constructs
allow a program to execute in parallel in a multicore computer. However, if
Cilk keywords are taken out from a Cilk program, the result is a syntactically
and semantically correct C version of it. This resulting program is called the
C-elision of the Cilk program.

In Cilk-1, the scheduler was an identifiable part of the code. In Cilk-5 both
the compiler and the runtime share the responsibility for scheduling. The per-
formance of a Cilk\footnote{From this point onwards, when referring to Cilk, it is a reference to Cilk-5.} computation can be characterized based on two parameters:
the work associated to the computation, which is the total time required to execute it serially, and the critical-path length, which is the execution time of the computation using an infinite number of processors.

A lot of the efficiency of Cilk comes from the application of the work-first principle [29] which states the following:

“Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path”.

In Cilk each of the workers keeps a ready queue to store threads ready for execution. The workers store the ready-to-execute threads at the tail of this queue. When the worker spawns a procedure, the corresponding thread is popped from the tail of the queue. When a ready-to-execute thread is stolen, the thief that makes the steal takes is from the head of the queue. This stealing strategy favors divide and conquer applications since the threads residing at the head of the queue are the ones that generate more work and further parallelism.

The Cilk language. In Cilk, the keywords spawn and sync are used in order to specify parallelism and synchronization respectively. In order to explain these extensions that Cilk incorporated to C to provide parallel execution support, consider the code shown in Figure 2.7. The keyword cilk in line 1 indicates that the function nfib is a Cilk procedure. Lines 7 and 8 contain the spawning of a couple of other Cilk procedures. The keyword spawn indicates that a spawned procedure can continue its execution in parallel with its child. The Cilk runtime schedules all the spawned procedures among the available CPUs in a parallel computer. In the referred code, the keyword sync acts as a local barrier that ensures that a Cilk procedure waits until all its spawned children return. This ensures that when the parent procedure executes the addition in line 10, the involved values will have been generated already, preserving in this way the correctness of the computation.

Cilk also provides more advanced constructs for handling parallelism. The keyword inlet can be used to define a C function that is internal to a Cilk procedure. In Cilk an inlet is defined in the declaration part of a procedure. It specifies the actions to be performed when a spawned procedure returns. The attractiveness of using inlets is that Cilk guarantees that they operate atomically related to other inlets when updating the variables of a given procedure’s frame. This provided implicit atomicity makes it easier to reason about concurrency
without using locks or the declaration of critical regions [71]. Cilk also provides
the keyword `abort` which is used to stop some parallel work that at some point
in a computation is discovered to be unnecessary. A typical example of this can
be a parallel search in which the solution has been found and it is not necessary
to explore the remaining possibilities.

```c
1. cilk int nfib(int n)
2. {
3.   if (n < 2)
4.     return 1;
5.   else {
6.     int x, y;
7.     x = spawn nfib(n - 1);
8.     y = spawn nfib(n - 2);
9.     sync;
10.    return (1 + x + y);
11.  }
12. }
```

Figure 2.7: Cilk code example of the `nfib` function.

**Cilk’s compilation strategy.** When a Cilk program is compiled, the Cilk comp-
piler creates two versions of each procedure. These are called the `fast clone` and
the `slow clone`. The `fast clone` runs whenever a procedure is spawned. However,
when a procedure is stolen by a `thief`, it is converted to a `slow clone`.

The `fast clone` is executed very similarly as its corresponding `C elision` does
and it provides minimal support for parallelism. Since this procedure version is
the one that is executed the vast majority of the times (when enough parallelism
is present), it is heavily optimized in order to provide a good performance. The
`slow clone` on the other hand, provides full support for parallelism. Whenever a
`slow clone` is executed, it has to restore the variable state of the stolen procedure.
Even though the execution of the `slow clone` generates a substantial overhead, its
contribution to work overhead is minimal since `slow clones` are rarely executed.

**Work stealing implementation.** One of the main components of Cilk is its
scheduler. As mentioned before it uses work stealing for achieving load balanc-
ing when executing a parallel computation. Most of the time the Cilk workers
(threads executing Cilk procedures) enqueue and dequeue `threads` without the
need of locking its queue, however when a `thief` and the `victim` try to pop the
same frame from the victim’s queue, there is a mechanism that guarantees that only one of them is able to get the frame. This mechanism is implemented using the THE protocol (acronym coming from Tail, Head and Exceptions) [29].

Before explaining the protocol, it must be mentioned that before trying to steal from a chosen victim, a thief has to acquire a lock L corresponding to the queue of the victim involved in the steal attempt. Therefore if there were multiple thieves trying to steal from the same victim at the same time, since only one can acquire the mentioned lock, only one of them would actually perform the steal attempt.

In the mentioned case in which a victim and a thief try to pop the same frame from the victim’s queue, the THE protocol guarantees that, if when trying to pop a frame, the thief detects that the victim already started the popping of the same frame, the thief stops its current attempt and retreats. This action is repeated until it detects that it is not competing with the victim. If on the other hand, when trying to pop a frame, the victim detects that a thief has already started the popping of the same frame, then the victim stops its attempt and proceeds to acquire the lock L (which in this case is held by the thief). The use of the THE protocol makes the Cilk work stealing mechanism highly efficient.

Intel Threading Building Blocks

The Threading Building Blocks (TBB) system [46] is a C++ template library designed to provide support for porting applications to multicore platforms. This library allows developers to write parallel applications without having to specify directly thread creation and management. In order to provide efficient performance and scalability, the library supports fine grain parallelism through the use of tasks. In TBB a task is a user level object whose scheduling is managed by the TBB task scheduler. TBB uses a similar mechanism to the one used in Cilk, in fact the Cilk scheduler inspired the design of the TBB task scheduler.

In TBB, a pool of native threads is maintained and also there are ready pools of tasks assigned on per-thread basis. At the initialization stage an appropriate number of threads are created in the pool by the TBB task scheduler. Then the ready pools of tasks are maintained using a randomized work stealing algorithm [13, 14]. In TBB the task scheduler is exposed to the programmer in the form of an API [46]. The above API includes the methods spawn, spawn_and_wait_for_all and spawn_root_and_wait which are used to perform the
spawning of tasks and to set the synchronization points in a parallel application. The TBB task scheduler uses a *Breath-First Theft and Depth-First Work* strategy when performing a parallel computation. This means that each worker executes the tasks at the tail of its local pool and, in the case of stealing, it takes tasks from the head of remote pools. This with the aim of getting a balance between efficient execution and parallelism. It is worth noticing that this strategy is also employed by the Cilk scheduler. However, one of the big differences between TBB and Cilk is that when a TBB worker gets to a synchronization point in order to wait for other tasks to finish, this worker is freed by the TBB task scheduler so that it is able to execute or steal other tasks while it waits [46]. Cilk on the other hand, as mentioned earlier, waits in the synchronization point until all the pending spawned procedures return [29, 14].

As mentioned earlier, TBB is a C++ template library which cannot rely on compiler specific transformations and modifications to provide correctness and performance. However in order to get better performance when building a parallel application using TBB, developers are able to perform some scheduling optimizations manually. The most important of these optimizations are the *continuation tasks*, *scheduler bypass* and *task recycling*.

**Continuation Tasks.** One of the problems that might arise when using the *Breath-First Theft and Depth-First Work* strategy is a stack overflow. This problem might happen when a TBB worker that reaches a synchronization point steals a remote task from other worker. In this case the subtree of the stolen task is unfolded on top of the waiting tree on the CPU stack. There might be the case in which the repetition of this scenario leads to a stack overflow.

Taking into account the above situation, the TBB task scheduler forces any worker that is waiting for other task to finish, to steal only tasks that are deeper than any waiting task. In this way the stack growth is limited however, this mechanism also reduces the number of possible tasks to be stolen and therefore it limits parallelism. This situation can be handled more efficiently using *continuation tasks*.

*Continuation tasks* define a mechanism that allows limited stack growth without restricting the number of stealable tasks. A *continuation task* can replace a normal task in the graph. When the normal task is replaced, its space in the CPU stack is freed, leaving in this way only the tasks that are actively executing (the
children of the replaced task) on the CPU stack. When all the children complete, the continuation task is then spawned to continue with the work of the previously replaced task.

**Scheduler Bypass.** Using this technique it is possible to reduce overheads produced by scheduling. When using scheduler bypass the corresponding execute function of a task [46] returns explicitly the next task to execute. The fact that the next task to be executed is already known avoids the execution of the more complex task selection logic in the scheduler.

**Task Recycling.** If a TBB application makes use of continuation tasks and scheduler bypass, it can also make use of task recycling. By default when a task returns, the associated task object is deallocated. Using task recycling it is possible to keep a task object alive beyond the return of its corresponding task, allowing in this way to assign that object to a different task. This object recycling avoids both, the deallocation of the object of the completed task, and the allocation of another object for the new task.

**OpenMP Tasks**

OpenMP [2] is an API that provides support for parallel programming in shared memory multicore computers. The programming languages it supports are C/C++ and Fortran. The OpenMP 3.0 [8] and later specifications include the OpenMP tasking model which allows the expression of task level parallelism using OpenMP constructs, a capability not provided in previous specifications.

The task construct is used to generated a new task, which in the OpenMP context is a piece of executable code with its associated data environment. It is similar to the Cilk frames [29, 14] or the TBB task objects [46]. A task can be executed by any CPU located in the current team\(^2\) [2, 8]. This execution can be immediate or be deferred for later, and also can be performed in parallel with the execution of other tasks.

The synchronization of tasks in OpenMP is done using the taskwait and taskgroup constructs. Similar to Cilk’s sync, the taskwait construct suspends the execution of a task until all its children tasks complete. On the other hand,\(^2\)

---

\(^2\)A team of threads is initialized using the parallel directive which is used to mark a region of code that is executed by multiple threads.
the taskgroup construct suspends the execution of a task until the completion of only the children tasks that were created in a given structured block [74].

In OpenMP 3.0 the programmer has the option of using tied tasks (the default in OpenMP) or untied tasks. The former are tasks that must be executed by the thread that creates them. The latter are tasks that can be executed either by the thread that initially creates them, or by any other thread in the current team.

The OpenMP 3.0 task scheduling systems designed by the OpenMP 3.0 tasking subcommittee are based on Nanos v4 [74]. Nanos v4 is library based on the nano-threads programming model [66], it was built using the POSIX threads standard [15]. Nanos itself is an implementation of a user level threads package. The library implements a global queue to store those nano-threads that any CPU in the system can execute. It also implements a set of local queues from which only the associated CPU can execute the corresponding stored nano-threads.

Different scheduling strategies were evaluated when designing the OpenMP 3.0 task scheduling system [23]. Two main approaches were evaluated in this study: Breath-First Scheduling and Work-First Scheduling.

- **Breath-First Scheduling.** In this scheduling system every task created by a thread is placed into a global pool, called the team pool, continuing with the execution of the parent task afterwards. In this way, in a recursive application, all the tasks corresponding to the same recursion level are created and then, the execution of the tasks of the next recursion level begins.

  When stored in the team pool, tasks can be executed by any thread in the current team. Once a task is suspended, if it is a tied task, it is placed in the local pool of the corresponding thread and, therefore only can be resumed by this thread. If the suspended task is an untied one, then is it placed in the team pool so that it can be resumed by any thread in the current team. A thread always searches first in its local pool for some task to execute, if it is empty, then it tries to get a task from the team pool.

- **Work-First Scheduling.** In this scheduling system, when a thread creates a task, the parent task is suspended and the executing thread switches to execute the new task. Suspended tasks are placed in the local pool of the owner thread. When a thread searching for a task to execute finds its local pool empty, it then tries to steal a task from other threads. The stealing
strategy is to search thread by thread starting with the next thread number. In this way thread 0 starts searching in thread 1, thread 1 in thread 2 and so on, so that thread n searches first in thread 0. This strategy is used with the aim of minimizing contention and is used as the default one.

Another stealing strategy was also implemented. In this, the running thread first tries to steal the parent task of the suspended task. In case it cannot be stolen (due to it is waiting in a synchronization point or it is already running), the default strategy is then applied. According to the OpenMP restrictions, once a task has become tied to a particular thread, it cannot be stolen\(^3\).

After evaluating the above scheduling systems it was found that when an application uses untied tasks, in general, the work-first scheduler’s performance is better than its breath-first counterpart. However the work-first scheduler’s performance degrades significantly when in an application there are no untied tasks. On the other hand, the breath-first scheduler’s performance does not change significantly when an application uses either tied or untied tasks.

At the end the OpenMP 3.0 specification does not describe the way in which task scheduling is done, therefore it is left to each particular implementation.

Other Software Scheduling Approaches

Since the appearance of Cilk [29, 14], other task parallel runtime systems have been implemented. One of these is Cilk++ [52], which is the evolution of Cilk that extends the C++ programming language for writing Cilk-like parallel applications. Cilk++ uses basically the same scheduling system as Cilk. Additionally it incorporates the \texttt{cilk} for construct for exploiting loop level parallelism; it also provides \texttt{reducer hyperobjects} to avoid data races in code in which non-local variables are accessed.

The Java Fork/Join Framework [49] is a task-parallel runtime system for Java. It was implemented taking as a model the Cilk runtime system [29, 14]. Having been designed to work with Java, its performance is sensitive to the JVM’s code optimizations available, and also to the garbage collection system [49].

SLAW [30] is a locality aware adaptive work-stealing scheduler supporting the Habanero-Java (HJ) language. HJ is a dynamic task parallel language based on

\(^3\)However tied tasks that has not been executed yet are eligible for being stolen.
Java and is derived from X10 v1.5 [18]. This scheduler applies the concepts of places, stack condition and fresh task threshold. These in combination with locality hints given to the runtime system either by the programmer or the compiler, make the scheduler to be locality-aware and to achieve adaptive scheduling. The locality awareness is achieved grouping the workers into places, whilst the adaptive scheduling is performed switching between the work-first and the help-first scheduling policies [30].

The Microsoft Task Parallel Library (TPL) [51] implements a runtime system that makes use of tasks and replicated tasks for implementing the parallel execution methods included in the library. One of the main sources of efficiency of this library is the case in which a running thread reaches a synchronization point, but one of the pending tasks has not been started yet. When this happens, the running thread executes the pending task directly. This mechanism is implemented using duplicated queues [51].

StackThreads/MP [73] is a library that implements a task parallel runtime system on top of Pthreads [15]. It uses one physical and one logical stack per worker in order to implement the scheduling system [73]. The links among the frames stored in the logical stack determine the computation flow and, these links may or may not match with the one determined by the frames resident in the physical stack. The arguments of the procedures are kept in the physical stacks of the workers.

Tflux [69] is a system that supports the Data-Driven Multithreaded (DDM) execution model [25]. Thread scheduling is performed by the Thread Synchronization Unit (TSU) which can be implemented using either software or hardware. DDM is an execution model which performs parallel execution using a dataflow-like scheduling of user level threads called DThreads. Tflux generates a Synchronization Graph to represent the dependencies among DThreads. This graph is loaded in the TSU before executing any DThread, then the scheduling is done dynamically at runtime in a data-driven fashion. The TFlux with a TSU implemented in software are TfluxSoft and TFluxCell [69].

2.5.2 Hardware Scheduling

When working with parallel applications, it is necessary to ensure that the cost of task migration is small compared with the computation associated with the migrated tasks. If this does not happen, then the communication overhead plays
a negative role in terms of performance. Software only schedulers perform communication actions using the shared memory. This has been demonstrated to be efficient in systems with low processor counts however, in large scale systems, the shared memory becomes an inefficient channel due to the high cost of scheduling operations performed throughout the memory hierarchy [48, 22].

Task generation and execution are other aspects that are affected by task granularity. When the tasks are very fine grain, the overheads associated with the saving of the state of each task, as well as the ones related to the enqueuing/dequeuing of tasks can become a very significant part of the execution time, affecting substantially the overall performance [29]. In order to address the above mentioned issues, it has been proposed to provide hardware support to the scheduling system in order to accelerate task creation, enqueuing, dequeuing and migration.

**Carbon**

In Carbon [48], it was proposed to use architectural support in order to exploit fine grain parallelism in parallel applications executed in a shared memory CMP. It uses a *Global Task Unit (GTU)* that implements hardware queues per thread for storing the tasks to be executed. Each thread keeps an unbounded local queue in software from which, a portion is stored in the GTU. The load balancing system is implemented also in hardware inside the GTU using work stealing techniques.

The whole system provides support for executing task and loop level parallel applications. One of the main points of Carbon is that most of the hardware required for its implementation is located far from the cores. This allows that only minor modifications are required inside the cores. These modifications are related to the implementation of the *Local Task Unit (LTU)* [48]. In the work presented in [48] the GTU is placed at the same level as the L2 caches so that the on-chip processor interconnect is used by all the cores in order to access it. Carbon implements instructions that allow the CPU to perform GTU related operations such as enqueue/dequeue operations for both, task level and loop level parallel applications. There are also instructions for handling the overflow and underflow of the hardware queues.

When a thread attempts to get a task from an empty queue, the thread is blocked. When all the threads are blocked, a special task to signal the end of the parallel region is sent by the GTU to each thread. When a queue in the GTU
gets full, a group of the resident tasks is backed up in a software queue. Similarly when a queue in the GTU gets lower than a defined threshold, a group of tasks is brought from the software queue.

A *Local Task Unit* is kept per core in order to perform task prefetching with the purpose of hiding the access latencies to the GTU. Carbon was capable of providing scalability in applications in which the tasks were on the order of hundreds of instructions.

One of the limitations of Carbon is that the scheduling policies provided by the system are completely hardwired in the GTU. This fact does not allow the exploration of different scheduling policies without hardware modifications.

**Asynchronous Direct Messages**

Asynchronous Direct Messages (ADM) [22] is an asynchronous communication mechanism that allows the implementation of fine grain scheduling systems. This proposal aims to get a balance between speed provided by dedicated hardware, and flexibility provided by software when implementing fine grain scheduling systems. Different from Carbon [48] that implements the load balancing system in hardware inside the *GTU*, ADM’s proposal comprises more sophisticated runtime systems at software level, which make use of low latency messages for load balancing. These runtime systems make use of minimal hardware support for their implementation. In this way ADM implements more flexible dynamic load balancing systems than the one provided in Carbon. This flexibility allows the use of different scheduling policies.

ADM are low latency messages sent and received directly in CPU registers. They avoid the scheduling system having to perform communication using the memory hierarchy. The messages are initiated by software threads and received synchronously or asynchronously using a receive instruction or a user level message handler respectively.

The hardware support for implementing ADM comprises a buffer for the received messages, a table for translating between *Thread IDs* and physical cores called the *Thread ID Translation Buffer*, a buffer for the messages to be sent and an Interrupt Controller/Generator. This hardware support is used by the software runtime systems in order to implement the scheduling policies aimed to exploit different kinds of parallelism such as task-level and loop-level [22].

ADM provides flexibility for exploring different scheduling policies by means
of implementing them in the ADM software runtime systems. This allows the evaluation of different alternatives in the search of performance when executing parallel applications. However, this flexibility comes tied to a higher complexity at the time of implementing the ADM-based runtime systems.

Multiple Instruction Stream Processor

The *Multiple Instruction Stream Processor (MISP)* [32] is a CMP architecture that introduced *sequencers*. A sequencer behaves like a *Multiple Instruction Multiple Data (MIMD)* functional unit. A MISP processor includes two or more sequencers. Only one of these is managed by the Operating System (OS) whilst the rest are managed by the applications. The MIPS architecture preserves the shared memory programming model.

The OS sees a MISP processor as a single logical CPU to which it can schedule OS threads for execution. The application managed sequencers execute a stream of instructions in parallel with other sequencers. These streams of instructions are user-level threads called *shreds* and belong to the same OS thread. In this way an OS thread can be composed of several *shreds*.

A sequencer is equivalent to a hardware thread. It is capable of fetching and executing instruction streams. There can exist asymmetry in terms of performance, functionality and/or power among the sequencers within a MISP processor. However, the OS does not have to deal with this asymmetry since the sequencers are managed by the application.

The MISP processor sequencers are accessed via the *ShredLib* runtime library. It implements shred control and synchronization functionality. It also provides equivalent functions to those found in the Pthreads [15] and Win32 Threads [9] APIs to facilitate the migration of multithreaded applications.

Network Driven Processor

The *Network Driven Processor (NDP)* [19] is a CMP architecture that proposed a different *hardware-software contract* in which there is an aggressive partition of applications with the aim of exposing all the potential parallelism. Once the parallelism is exposed, hardware mechanisms are used for exploiting it.

The NDP system provides an API that allows the creation of threads exposing their characteristics to the underlying hardware. It then performs the scheduling
of the created threads into the cores releasing the programmer/compiler from dealing with it.

In NDP, thread creation is performed directly in hardware. It also supports runtime thread-cloning which allows it to efficiently execute aggressive do-all and stream parallelism. The data flow of the applications is kept using queues. These queues are used for exposing inter-thread communication, information used by the hardware to handle network bandwidth appropriately. Critical thread dependencies are also communicated by the above queues; these dependencies are used to keep record of the critical path of applications, as well as the data required for performing prefetching before thread execution.

A NDP node comprises the thread table which stores thread information, the queue table that stores information related to the producer-consumer data flow queues between threads, and the tile table which stores information related to the workload of the tiles in the entire chip. The mentioned tables are backed up in software. There is also a scheduler which maps threads into cores. The NDP network is designed to provide fast access to thread scheduling information such as the one stored in the tile table, which is updated by the network. One of the main points of the NDP system is that the NDP scheduler is the one responsible for initiating execution in the processor core.

Look-Ahead Task Management Unit

The Look-Ahead Task Management Unit (TMU) [68] is a hardware system aimed to solve dependencies between tasks at runtime. The TMU was designed for accelerating the execution of task parallel applications in which there are dependencies among the tasks, such as H.264 parallel video decoding with macro-block level parallelism [75]. The purpose of the TMU is to release the cores from the execution of the task management computation of the applications. This computation solves the dependencies between tasks, determining which of these are ready for execution.

On a 16-core CMP described in [68], each TMU is connected to four cores, as well as to a centralized task queue. While the cores are executing tasks, the TMU computes the task dependencies for identifying tasks ready to be executed. Once a core finishes the execution of a tasks, it sends a signal to the TMU and, if there is a task ready for execution, this is sent to the core by the TMU.

Task queues are used for distributing ready to execute tasks among the TMU’s.
In this way, a TMU that does not find ready to execute tasks coming from the tasks computed on the cores it is connected to, still could send a task to one of its cores in case of getting a signal asking for a task.

Each core has a dedicated hardware data structure in which a list of candidate tasks is stored. Each entry in this list contains information about a task, its arguments, its dependencies, a look-ahead function [68] pointer for the TMU, and status flags. This list can be read by the CPU when searching for ready tasks. Also the above list can be accessed by the TMU for updating tasks status, checking the appropriate look-ahead function to execute, as well as for updating the status of tasks.

**Hardware Task Scheduler**

The *Hardware Task Scheduler (HTS)* [6] performs task management operations in hardware for accelerating parallel applications that exhibit a repetitive inter-task dependency pattern. For doing this, the application must provide information such as dependency pattern, as well as loop boundaries with which it is possible to build a dependency graph. With this information, the HTS sets tasks for execution in the different cores and, at the same time builds gradually the dependency graph. In this scheme the cores keep asking for tasks, sending a signal to the HTS when each task is finished. When the entire graph has been executed, the HTS signals this to all the participating cores.

The HTS is connected to several homogeneous cores in a shared memory multiprocessor System-on-Chip. One of these cores configures the HTS with the corresponding dependency pattern and loop boundaries, this is named the *master* core. The other cores keep requesting tasks from the HTS, these are named *slave* cores.

The HTS hardware comprises the following elements: several *slave ports* to be connected to several cores, as well as to other HTS’s. There is a *master port* used for performing task stealing from other HTS’s. The *control unit* handles HTS operation. A *Floating Ready Task FIFO* queue stores ready tasks that have not been assigned to any slave port. *Slave Ready Task FIFOs* are used to store ready tasks assigned to slave ports. The *Slave Candidates Buffer* is used for storing candidates of a task that is being executed. The *Synchronization Buffer* is used to store the coordinates of the finished tasks. The Repetitive Dependency Pattern Buffer stores the X and Y coordinates relative to the repetitive dependency
TFlux with a Hardware TSU implementation

In TFlux, as mentioned in 2.5.1, the Thread Synchronization Unit can be implemented in hardware as is the case of TFluxHard [69, 25]. The TSU comprises three units: the Thread Issue Unit (TIU), the Post Processing Unit (PPU) and the Network Interface Unit (NIU). The function of the PPU is to check when a DThread becomes ready for execution and, when this happens it sends it to the TIU. As expected, the work of the TIU is to schedule and prefetch DThreads identified as ready for execution by the PPU. The NIU deals with the communication between the TSU and the interconnection network.

2.6 Summary

This chapter has presented the concepts related to the work developed in this thesis. It has provided a description of dynamic load balancing, work stealing, active messages and lazy task creation.

It has also provided the reader with a survey of the dynamic scheduling systems which are relevant to the work developed in this thesis. All these systems are aimed at exploiting parallelism from applications executed on shared memory Chip Multiprocessors. The descriptions of software-only and hardware-assisted systems were presented. The first were created due to the need for having tools that facilitate the development of parallel applications without dealing directly with thread creation and synchronization. The latter were created due to the need of exploiting fine grain parallelism efficiently, something not achievable using software-only systems.

The next chapter will describe the Spreading Load with Active Messages (SLAM) system [81], including the way in which it exploits parallelism using work stealing, active messages and lazy task creation techniques.
Chapter 3

SLAM (Spreading Load with Active Messages)

3.1 Introduction to SLAM

This chapter provides a description of SLAM (Spreading Load with Active Messages), a parallel programming model based on Work Stealing and Lazy task creation for performing dynamic load balancing. This description includes the elements of the model as well as the way it performs serial and parallel computations. In order to illustrate how SLAM works an example is provided going from a high level description to a more detailed one. This detailed description explains how SLAM exploits parallelism at runtime, as well as the programming interface of the model.

3.1.1 SLAM

SLAM (Spreading Load with Active Messages) [81] is a highly dynamic parallel computing model. It is designed to distribute efficiently the workload among multiple CPUs at run time. The model is appropriate when the workload expands and shrinks dynamically and, as a consequence static partitioning is not possible. It makes use of Work Stealing [13] and Lazy Task Creation (LTC) [60] to achieve dynamic load balancing. The communication among the different processing nodes is performed using a customization of the Active Messages [76] approach.

In order to take advantage of SLAM when writing parallel programs, the problem must be structured in a functional way. If each function has a private
set of arguments and writes the produced result to a private memory location, then several functions can be executed in parallel as long as their arguments are available.

Once a program has been structured in a functional way, the programmer specifies which tasks can be executed in parallel and the runtime system deals with the load balancing and synchronization issues automatically.

3.1.2 SLAM Task Stack

SLAM makes use of distributed queues for storing the tasks to be executed. There is one queue attached to each core in the system in the form of a Task Stack (TS); each TS is private to the core it belongs to. The structure of a TS entry comprises 3 pointers; there is one pointer for each of the following elements:

- The function to be executed.
- The arguments to be used by the corresponding function.
- The address where the produced result is required.

In order to access the task that has to be executed each time, there is a pointer that points to the appropriate TS entry, the Task Stack Pointer (TSP). Once each task is finished, the TSP is updated to point to the entry containing the next function to be executed and so on until the computation is finished. In this way the tasks stored in the TS are executed once they are popped by the owner CPU; a task being executed can push new tasks onto the TS or modify an existing one. This behavior makes the TS a key element for SLAM. The structure of the TS is shown in Figure 3.1.

Stealable and non-Stealable Tasks

In SLAM there are two types of tasks defined: Stealable and non-Stealable. Stealable tasks are those that can be either executed by the owner CPU or that could be stolen and executed by a remote one. non-Stealable tasks are those that must be executed by the owner CPU. This latter type includes tasks used to perform synchronization and load balancing operations. All the tasks are pushed by the owner CPU on the TS and are popped as the execution proceeds.
CHAPTER 3. SLAM (SPREADING LOAD WITH ACTIVE MESSAGES) 56

3.1.3 Load Balancing

SLAM achieves load balancing using work stealing [13]. As mentioned before, each CPU keeps a pool of tasks to be executed in its TS. When any of the CPUs runs out of work, it executes a Steal task. This task selects a victim CPU from which to steal; thereafter it sends a message requesting a task to the chosen victim.

Each CPU in the system has an interrupt driven communication mechanism to deal with remote requests asking for work. When the communication mechanism of a victim gets the request, it serves it either taking a task from its local pool and sending it to the requesting CPU, or sending a refusal in the case where there are no stealable tasks available.

The migration of a task from the local pool of the victim CPU to that of the requesting CPU is very simple. It basically involves copying the entry corresponding to the stolen task from the TS in the victim to the TS in the requesting CPU. There is also additional information that is sent to the requesting CPU; this information is related to the place where the result of the stolen task must be returned, as well as the memory address of the corresponding TS entry.

3.1.4 Active Messages

The original idea of Active Messages [76] is that each message sent through the network has, in its header, the address of a user level handler. This handler is executed on arrival. This user level handler is executed by the receiving CPU. The function of this handler is to extract the message from the network and to insert it into the ongoing computation of the CPU. The argument of the message
is its own body. These messages were used to perform communication among CPUs in a multiprocessor system with the objective of hiding the communication cost among CPUs.

### 3.1.5 SLAM Network

In order to establish communication among the CPUs in the system, the existence of a NoC (Network on Chip) is assumed. This network is used to carry messages related to dynamic load balancing activities. These messages can be:

- The request for a task originated when a Steal\(^1\) task is executed. This kind of message is generated in two cases. First, when a CPU runs out of work. Second when a CPU had a task stolen and it reaches the point in the execution where that result is required, but it is not ready yet. In both cases the CPU generates a request for a task to a remote CPU.

- The answer to a remote request for a task. These messages are generated in response to a remote request from a thief CPU which is looking for some work to do. The answer for this kind of request can be either a task exported to the thief CPU, or a refusal because there is no exportable task available in the victim CPU.

- The notification of a stolen task completion. This kind of message is generated when a CPU finishes the execution of a task that was previously stolen. This is to indicate to the corresponding victim that the result of the stolen task is ready. In this way if the victim needs the result of the stolen task for another computation, it can proceed safely.

### 3.1.6 SLAM Communication Control

The Communication Control (CC) is the controller that synchronizes all the communication related to dynamic load balancing in SLAM. It takes control of the local TS each time it has to deal with the requests for tasks going to and coming from remote CPUs. It also handles the generation and receiving of notifications related to the completions of stolen tasks.

In SLAM the CC is an interrupt driven communication mechanism. The fact that it is interrupt driven eliminates the possibility of race conditions, since all

---

\(^1\)The meaning of this task is explained in Section 3.2.1
the activities involving access to SLAM hardware are performed privately by the
owner processing node.

**Communication Control Operations**

The CC deals with three main operations:

- **Issuing of requests for work.** This involves selecting a victim CPU from
  which to steal a task, and after the selection takes place, to insert the
  appropriate message in the SLAM network. After this is done, the CC has
  to wait for the response from the victim CPU in order to decide the next
  action to perform.

- **Handling responses to requests for work.** In this case the CC has to deal
  either with a refusal from the victim CPU or with the receipt of a stolen
  task. In the case that the response is a refusal to a request for work, the
  CC has to repeat the process of selecting a victim CPU and sending the
  appropriate message through the SLAM network. In the case that the
  response is a task sent by the victim CPU, the CC has to push the stolen
  task on the TS of the thief CPU so that it can be executed in the same way
  the victim would have done it.

- **Serving requests for work.** When there is a remote request for work, the CC
  has to check whether there is an available task to be sent to the thief CPU.
  In the case that there is a task available for exporting, the CC extracts
  it from the TS of the victim and sends it to the thief through the SLAM
  network. In the case that there is no task for exporting, then the CC simply
  sends a message to the thief indicating a refusal to the request for work.
3.2 How SLAM Works

3.2.1 Evaluation of $Nfib(n)$

In order to illustrate how SLAM works, a very simple example is provided. The $Nfib$ function is defined as:

$$Nfib(n) = 1, \text{ if } n < 2$$

$$Nfib(n) = 1 + Nfib(n-1) + Nfib(n-2), \text{ if } n \geq 2$$

This example, although very simple, is very illustrative as it clearly shows some of the operations that SLAM performs when doing a computation. This problem contains functions that get a set of arguments and return a produced result storing it in a specific memory location. It also has synchronization points that must be respected in order to preserve correctness when doing the computation in parallel.

Serial Evaluation

The evaluation of $Nfib(5)$ in a single core machine is shown in Figure 3.3. As shown in Figure 3.1 the first field of each task descriptor is the address of a piece of code which will perform the appropriate evaluation. Figure 3.2 shows the code executed for each evaluation of $Nfib(n)$ and together with Figure 3.3, provides a clear view of the operations required.

Figure 3.3 shows the TS0 corresponding to CPU0 in a single core machine. The entry numbers are shown at the right hand side of the TS. The arguments array corresponding to CPU0 is also shown. The arguments array is an array of data structures, each one containing two values for this case (the number of values can be adjusted depending on the number of arguments required to compute different kind of problems). It must be noticed that each element in the arguments array has a one to one correspondence with a TS entry. In this way, the $i^{th}$ element of the arguments array of a certain CPU corresponds to the $i^{th}$ entry of the TS belonging to that same CPU.

As mentioned earlier, each task descriptor has three fields. These fields can be observed in the Task Stack 0 in Figure 3.3. The first one contains the address of the function to be executed (either $Nfib$ or $Add$). The second field contains
CHAPTER 3. SLAM (SPREADING LOAD WITH ACTIVE MESSAGES)  60

a pointer to an entry in the arguments array. The last one holds also a pointer, but to an individual value in the argument structure, that is the place to which the value is to be returned. In Figure 3.3, the following notation is used:

- $Am[n]$ is a pointer to the $n^{th}$ element of the arguments array for processor $m$.
- $Am[n].i$ is the $i^{th}$ value of the $Am[n]$ element.
- $&Am[n].i$ is the memory address of the $i^{th}$ value of the $Am[n]$ element.
- $&TSm.i$ is the memory address of the $i^{th}$ entry of the Task Stack $m$.

```
1. void Nfib(int x) {
2.   if (x < 2) {
3.       return_value 1;  // store the result
4.       DEC_TSP;         // decrement TSP
5.   } else{
6.       OVERWRITE(Add,NS);  // overwrite Nfib with Add
7.       PUSH(Nfib,x-1,&Add_Args->a1,ST);  // push Nfib(x-1)
8.       PUSH(Nfib,x-2,&Add_Args->a2,ST);  // push Nfib(x-2)
9.   } 
10.  JMP_TSP;  // jump to the next task
11. }
12. 
13. void Add(int a1, int a2) {
14.   return_value = a1 + a2 + 1;  // store the result
15.   DEC_TSP;  // decrement TSP
16.   JMP_TSP;  // jump to the next task
17. }
```

Figure 3.2: Code of the $Nfib(n)$ function.

At the beginning the situation is the one shown in Figure 3.3(a). The code for $Nfib$ evaluates the first argument contained in the task descriptor (in this case 5) and detects that it is bigger than 2, so the main body of the code is executed. This creates a task $Add$ which will perform the final addition ($OVERWRITE$ is assumed to write the stack at the current TSP position) followed by two tasks to perform the recursive calls with the appropriate arguments created from $x$ (see lines 7 and 8 in Figure 3.2). These tasks have pointers ($&Add_Args->a1$ and $&Add_Args->a2$) indicating where the returned values are required ($PUSH$ will increment the TSP and then write the fields of the task descriptor in the TS). At the end of the $Nfib$ code, the situation is that shown in Figure 3.3(b), and the final action is to jump directly to the last pushed $Nfib$ task whose argument is 3. The
process is repeated creating the situation shown in Figure 3.3(c). The next execution of $Nfib$ has an argument 1, which is smaller than 2. In this case, the value 1 is returned to its corresponding address and the TSP is decremented resulting the situation shown in Figure 3.3(d). The result produced by $Nfib(1)$ is highlighted in TS0. This process continues with the appropriate actions performed by the $Nfib$ and $Add$ functions until no tasks are left.

There are some points to note about SLAM at this point:

- Normal serial scheduling is achieved by simply an indirect jump via the task stack at the end of each task.
- Both, the inline tasks and the (potentially) exportable tasks are held in the same form in the same task structure. This allows a very simple unified control structure for both, local scheduling and task exportation.

Parallel Evaluation

Figure 3.4 shows the TS0 and TS1 corresponding to CPU0 and CPU1 respectively in a dual core machine. The same as in Figure 3.3, entry numbers are shown at
Figure 3.4: \( Nfib(5) \) computed in parallel in a dual core machine using SLAM.

The right hand side of each TS. The arguments array corresponding to CPU0 is also shown (for simplicity the one corresponding to CPU1 is omitted).

It is assumed that the execution starts with an entry on TS0 which contains \( Nfib(5) \) although this is not shown in the diagram.

Figure 3.4(a) shows the state of the evaluation reached in Figure 3.3(c). TS1 contains a single entry which is a Steal task.

In general, packets are marked as Stealable only if they are likely to generate further parallelism. In this example, therefore, only the \( Nfib \) calls are so marked. As mentioned earlier in Section 3.1.1, the programmer is responsible of identifying and marking the potentially parallel tasks in an application. Also it must be noticed that the non-Stealable tasks preserve the correctness of a parallel computation, since these play the role of the synchronization points during execution.

CPU1 executes the Steal task at the top of its stack. Its function is first to select a victim CPU (CPU0 in this case) from which a task can be stolen and send a request for a task to the chosen victim.

Figure 3.4(b) shows the state after that CPU1 has executed its Steal task and successfully obtained an executable task from CPU0. This has resulted in 2 new tasks being pushed on top of TS1, Return and \( Nfib(4) \). These tasks have been
sent by CPU0 in response to the steal request made by CPU1.

The arguments for the stolen $Nfib$ task still exist only in the argument structure for CPU0 and will be accessed from there by CPU1. However, any new tasks generated by CPU1 will have their arguments placed in the corresponding structure for that processor. Note that the $Steal$ task is preserved on TS1 so that the steal process can be repeated in the future if necessary. It also can be observed that the stolen task has been overwritten with another $Steal$ task in TS0 of CPU0, which in this case, causes the victim to try to steal a task from another CPU if the result of the stolen task is not yet ready. It is worth noticing that trying to steal when the result of a previously stolen task is not ready, can potentially lead to a faster distribution of the workload in the system; this assuming that there is enough parallelism available in the computation being performed.

In Figure 3.4(c) CPU1 and CPU0 have finished the execution of $Nfib(4)$ and $Nfib(1)$ respectively and the results have been stored in the appropriate memory locations. The results produced by each task executed are highlighted in TS0.

The purpose of the $Return$ task is to communicate to the victim CPU the completion of a stolen task. In this example, when the stolen task is completed, CPU1 sends a $Null$ task to CPU0. Then the $Steal$ task that replaced the previously stolen task in TS0, is in turn overwritten by the $Null$ task sent by CPU1. When a CPU executes a $Null$ task it simply jumps directly to the next TS entry.

Finally in Figure 3.4(d) the $Steal$ task has been overwritten with an $Null$ task letting the victim know that the stolen task has been completed. When CPU0 executes the $Null$ task, it will only jump to the next TS entry.

It should be noted that, since the argument structures exist in main memory, they can be accessed by any CPU at any time. However, this will only occur for a task which has just been exported. In general, the argument structures will only be accessed by a single CPU and, assuming they are cached, will not place significant demands on any coherence mechanism.

In the same way as other stealing systems, if no parallel resources are available, serial execution would occur on the processor holding the initial call. Although the data structures are slightly more complex, the resulting stack based evaluation would be little different from that achieved by a conventional approach.
Finding a task for exporting

When getting a remote request from a thief CPU, the victim has to check whether there is an available task for being exported. This checking starts from the bottom of the stack to its top, so that if a task can be exported, it would be the oldest stealable one stored in the TS. This behavior favours divide and conquer applications since the oldest tasks are likely to generate a larger number of tasks, keeping the thief CPUs doing useful computation longer and hence reducing the time spent in communication activities. In the case that there is no task available for being exported, the victim CPU sends a refusal to the thief and then proceeds with the work it was doing before the request.

Marking tasks as Stealable and non-Stealable

As was mentioned before, tasks marked as Stealable are the ones that are likely to generate bigger amounts of parallelism. The Stealable tasks are marked by the programmer once these have been identified as potentially parallel tasks. The non-Stealable tasks are preserved in the TS of the owner CPU in order to preserve the correctness of the computations performed. In the example shown in Figure 3.2, it can be observed that the last argument of the PUSH statements is ST (lines 7 and 8), indicating that the corresponding tasks are Stealable. Similarly the last argument of the OVERWRITE statement is NS, which means that the corresponding task is non-Stealable.

It is the responsibility of the programmer to identify those tasks that can be executed in parallel. Once potentially parallel tasks have been identified, these must be marked so that the runtime system can take the appropriate actions in order to exploit parallelism automatically.

3.3 Programming with SLAM

In order to take advantage of the infrastructure of SLAM, programmers need to have knowledge of the architecture of the system. In particular, they must be aware of the stack architecture. There is a set of operations that SLAM uses in order to manipulate the TS and in turn, to be able to exploit parallelism and to perform load balancing. These operations are translated into functions that allow interacting with the infrastructure provided by SLAM.
CHAPTER 3. SLAM (SPREADING LOAD WITH ACTIVE MESSAGES)  65

3.3.1 SLAM Programming Interface

The list of functions available is presented below:

- **ARGS_ADDR**: returns the address of the location of the arguments of the function stored in the current entry of the TS.
- **RET_ADDR**: returns the address where the result produced by the function in the current entry of the TS is required.
- **OVERWRITE**: overwrites the current entry of the TS with some other data.
- **PUSH**: pushes a new task descriptor in the TS.
- **NEXT_TASK**: jumps to the address pointed by the Function pointer of the current TS entry.
- **DEC_TSP**: makes the TSP to point to the previous entry in the TS.
- **INC_TSP**: makes the TSP to point to the next entry in the TS.
- **PUSH REM**: pushes a new task descriptor in the TS of a remote CPU.

The TS provides the programmer with a structure that allows the division of a program into functions. The purpose of the TS is to create (whenever possible) a separate context with private arguments and return address for each function. Besides dividing a program into functions, as mentioned earlier, the programmer has also to define which ones could eventually be executed in parallel. The idea behind structuring programs in a functional way is to exploit the implicit parallelism that can be exposed using pure functions.

In order to illustrate how the TS is manipulated, Figure 3.5 shows a possible implementation for the $Nfib(n)$ function. It can be observed that SLAM code is very similar to conventional imperative programming. SLAM functions are simply tools to provide access to the TS hardware. However, the addition of the TS imposes the need to perform specific actions for manipulating the TS for both reading and writing information. These actions can be summarized in the following three steps:

1. Reading information from the current TS entry which could be the arguments or the return address pointers.
2. Building the arguments in the corresponding data structure for every new task. Since the only reference to the arguments in a TS entry is a pointer, these must be built before pushing a new task in the TS.

3. Pushing every new task in the TS.

Popping a task is an implicit operation performed each time the CPU jumps to the address pointed by the TSP for executing the next task.

```c
1. void Nfib() {
2.     task_args *p = ARGS_ADDR;  // get arguments address
3.     if (p->a1 < 2) {
4.         int *ret_ptr = RET_ADDR;  // get return address
5.         *ret_ptr = 1;           // store the result
6.         DEC_TSP;              // decrement the TSP
7.     } else {
8.         OVERWRITE(Add, NS);     // overwrite Nfib with Add
9.         (p+1)->a1 = p->a1 - 1;   // set argument of Nfib
10.        PUSH(Nfib, p+1, &p->a1, ST);  // push Nfib in the TS
11.       (p+2)->a1 = p->a1 - 2;
12.       PUSH(Nfib, p+2, &p->a2, ST);
13.     }  // jump where TSP points
14.     JMP_TSP;
15. }
16.  }
17. void Add() {
18.     task_args* p = ARGS_ADDR;  // get arguments address
19.     int *ret_ptr = RET_ADDR;  // get return address
20.     *ret_ptr = p->a1 + p->a2 + 1; // store the result
21.     DEC_TSP;                   // decrement the TSP
22.     JMP_TSP;                   // jump where TSP points
23. }
24. }
```

Figure 3.5: Code of the $Nfib(n)$ function using SLAM functions.

The above mentioned steps can be seen in Figure 3.5. The reading of information from the TS is done in lines 2, 5, 21 and 22. The building of arguments is in lines 11 and 14. The pushing of new tasks is performed in lines 12 and 15. It should be noted that the arguments are built in main memory in previously reserved locations. The locations where the results are stored are also reserved in advance. For example in the $PUSH$ statements, the return addresses are $\&p->a1$ and $\&p->a2$ which are variables located in the arguments data structure of the previously created $Add$ function. In the same way, the arguments data structures corresponding to the first and second $Nfib$ functions pushed are pointed to by $p+1$ and $p+2$ respectively.
There is a cost associated with the creation of a new task since it involves the creation of the arguments and the TS manipulation. On the other hand, since there is only a direct jump between the execution of one task and the next, there are minimal TS manipulation costs at this point.

3.4 Analyzing SLAM Code

In order to make a deeper analysis of the way SLAM is used for writing parallel applications, the code shown in Figure 3.5 is going to be used.

As mentioned in the previous section, there are three steps to follow when programming with SLAM. The step related to reading information from the TS can be seen in lines 2, 5, 21 and 22. Lines 2 and 21 show each an access to the TS for getting the address of the CPU’s arguments array element for the function resident in the current TS entry. Similarly lines 5 and 22 show reads of TS data in order to get the address where the result produced by each function must be stored.

The building of the arguments is done in lines 11 and 14. It can be seen that the argument for the next function is generated before the function is pushed in the TS. There are a couple of details to notice at this point. First, in order to build the arguments for the next task to be pushed, the proper element in the CPU’s arguments array must be accessed. Second, this element is easily reached because it is the next one following the element corresponding to the current function and hence to the current TS entry at any time. Line 11 shows a value being written in $(p+1)->a1$. Here, $p$ is a pointer to the CPU’s arguments array element corresponding to the current TS entry (and to the current function) so that $p+1$ is a pointer to the next CPU’s arguments array element corresponding to the next TS entry, where the new created Nfib function is pushed in line 12. In a similar way $p+2$ is a pointer to the CPU’s arguments array element corresponding to the second Nfib being pushed in line 15. This simple handling of a CPU’s arguments array is possible due to the stack structure provided by SLAM.

The step related to push every new task created in the TS is seen in lines 12 and 15. These new created Nfib functions are marked as Stealable by means of setting the last argument of each PUSH call to ST. Following that same line it can be seen that the Add function in line 9 with the OVERWRITE call is marked
as non-Stealable setting the last argument of this call to $NS$.

### 3.5 Hiding SLAM Low Level Operations

Exposing programmers to SLAM functionality using the functions for manipulating the TS gives them tools for writing parallel applications taking advantage of SLAM’s infrastructure. However, the programming style of the example shown in Figure 3.5 is not very friendly. This is mainly because there are low level operations exposed to the programmer. Operations like $ARGS\_ADDR$, $RET\_ADDR$, $JMP\_TSP$, $INC\_TSP$ and $DEC\_TSP$ can be automated from the source code doing some preprocessing and providing the appropriate compiler support.

#### 3.5.1 A Simplified SLAM Programming Interface

The code in Figure 3.6 shows a friendlier programming interface for SLAM. In this code, operations like extracting arguments, getting the return address of each function and decrementing the TSP are performed automatically. This releases the programmer from doing it manually. The building of the arguments is also simplified by packing it inside the $PUSH$ calls. $CONTINUATION$ and $NEXT\_TASK$ have replaced the calls to $OVERWRITE$ and $JMP\_TSP$ respectively.

**Simplified code VS Code with low level operations**

A comparison between the simplified $Nfib(n)$ code and the one including the low level operations is shown in Figure 3.7. The code including SLAM low level operations is shown in Figure 3.7(b) whilst the simplified code is shown in Figure 3.7(a). It can observed the way in which SLAM operations that have a well defined pattern are simplified. The coloured elements next to some lines of code delimit sections. Lines marked with a colour in Figure 3.7(b) are equivalent to lines marked with that same color in Figure 3.7(a).
CHAPTER 3. SLAM (SPREADING LOAD WITH ACTIVE MESSAGES) 69

1. void Nfib(int n) {
2. if (n < 2) {
3. RET_VAL 1;
4. } else{
5. CONTINUATION(Add);
6. PUSH(Nfib,n-1,&x,ST);
7. PUSH(Nfib,n-2,&y,ST);
8. }
9. NEXT_TASK;
10. }
11.
12. void Add(int x, int y) {
13. RET_VAL x + y + 1;
14. NEXT_TASK;
15. }

Figure 3.6: Nfib(n) simplified code assuming SLAM low level operations automated.

1. void Nfib(int n) {
2. if (n < 2) {
3. RET_VAL 1;
4. } else{
5. CONTINUATION(Add);
6. PUSH(Nfib,n-1,&x,ST);
7. PUSH(Nfib,n-2,&y,ST);
8. }
9. NEXT_TASK;
10. }
11.
12. void Add(int x, int y) {
13. RET_VAL x + y + 1;
14. NEXT_TASK;
15. }

1. void Nfib() {
2. task_args *p = ARGS_ADDR;
3. 
4. if (p->al < 2) {
5. int *ret_ptr = RET_ADDR;
6. *ret_pointer = 1;
7. DEC_TSP;
8. } else{
9. OVERWRITE(Add,NS);
10. 
11. (p+1)->al = p->al - 1;
12. PUSH(Nfib,p+1,sp->a1,ST);
13. 
14. (p+2)->al = p->al - 2;
15. PUSH(Nfib,p+2,sp->a2,ST);
16. }
17. JMP_TSP;
18. }
19.
20. void Add() {
21. task_args* p = ARGS_ADDR;
22. int *ret_ptr = RET_ADDR;
23. *ret_ptr = p->al+p->a2+1;
24. DEC_TSP;
25. JMP_TSP;
26. }

Figure 3.7: Comparison between Nfib(n) simplified code and Nfib(n) code that includes SLAM low level operations.
3.6 Summary

This chapter has provided a description of SLAM, its elements and the way parallel applications can be written using its functionality. SLAM is a general purpose highly parallel programming model for exploiting parallelism at runtime. It is particularly appropriate when the workload of applications expands and shrinks at runtime and in consequence, static partitioning is not a good choice. It makes use of Work Stealing [13] and Lazy Task Creation [60] for performing dynamic load balancing. In order to write parallel applications using SLAM the program must be divided into functions. Once it has been done, these functions can be executed using the TS and the runtime system can exploit parallelism dynamically.

The next chapter explains the implementation of SLAM developed in this thesis. It describes the way in which SLAM was adapted using relatively simple architectural support to work in a shared memory Chip Multiprocessor Architecture.
Chapter 4

SLAM Implementation

4.1 Introduction

The previous chapter presented a detailed description of SLAM at a conceptual level, the infrastructure it requires and how it is used for exploiting parallelism. It also presented a programming interface used for writing parallel applications using SLAM.

This chapter explains the implementation of SLAM performed in this work, how the Task Stacks (TSs) were implemented and how the Stealable and non-Stealable tasks are handled at hardware level. A description of the ISA extensions used in SLAM is also presented.

The chapter continues with a description of the actions taken by the Communication Control (CC) when performing dynamic load balancing operations. It is also explained the way in which having the CC attached to each CPU in the system avoids race conditions on the data stored in the TSs. At the end, the configuration of a Chip Multiprocessor architecture with SLAM hardware support implemented is described.

4.2 Task Stack

The task stack (TS) is where the tasks to be executed are stored. There is one task stored per TS entry. In order to execute a task, the owner CPU needs to pop it through accessing the corresponding entry in the TS.
4.2.1 Task Stack Pointer

The Task Stack Pointer is a register that is used in order to get access to each of the entries in the TS. This is a register that holds the value of the TS entry that has to be accessed by the CPU each time.

4.2.2 Task Stack Entry

Each entry in the TS contains the following information:

1. The *Function pointer*. Points to the function to be executed.
2. The *Arguments pointer*. Points to the arguments used by the function.
3. The *Return pointer*. Points to the address where the result produced by the function must be stored.

**Function Pointer**

When performing a computation using SLAM, a CPU updates regularly the TSP in order to keep control of the order of the tasks to be executed. The CPU jumps to the address stored in the *Function pointer* resident in the TS entry pointed by the TSP. While executing the task, the CPU also updates the TSP appropriately. Once the task is finished, the CPU jumps to the next task contained in the TS entry pointed by the TSP and so on, until the computation is finished.

**Arguments Pointer**

The handling of the arguments corresponding to each function is different from the conventional approach when using SLAM. Instead of using the *normal* stack to pass arguments to a function, these are created in a separate region in the main memory and referenced by the *Arguments pointer* stored in the function’s TS entry. In terms of memory accesses, this is slightly less efficient than implementing a function call on a conventional stack as there is an indirection required to access the arguments. It is also not obvious how the optimization of passing arguments in registers can be implemented as they must be accessible for reading and writing by exported tasks. The major reason for splitting the arguments from the information in the Task Stack is to allow for easy optimization of TS
manipulation, allowing in this way a simple implementation of stack caching as will be described later in this chapter.

**Return Pointer**

When a SLAM function returns a result, it writes it in the memory address referenced by the *Return pointer* of the corresponding TS entry. Since the address stored in the *Return pointer* is a memory location resident in main memory, it is accessible to all the CPUs in the system. This allows the result of a *Stealable* task to be generated either by its owner CPU, or by a CPU that previously stole it. Since the result is expected to be written in a unique memory location, the correctness of the corresponding computation is preserved.

### 4.2.3 Task Stack Operation

**Pushing a task**

When a task is pushed in the TS, it is simply written in the TS entry pointed by the TSP. Before this writing takes place, the TSP is incremented by 12 (12 bytes is the size of a TS entry) so that it points to the next TS entry. Once the TSP is pointing to the proper TS entry, the *Function*, *Arguments* and *Return pointers* are filled with the appropriate values. This process is executed every time a new task is pushed in the TS.

**Popping a task**

Popping a task from the TS is an implicit operation performed each time the CPU jumps to the address pointed by the *Function pointer* in the corresponding TS entry. When popping a task, there is no immediate updating of the TSP. Instead, this updating takes place inside the function being executed in the corresponding TS entry. Depending on the computation done inside the function, the value stored in the TSP can be incremented or decremented.

**Overwriting the Function Pointer**

Sometimes in the execution of a program that makes use of SLAM it is necessary to overwrite the *Function pointer* of the current TS entry. This is to create a new task that will wait and if necessary to compute the results provided by another
created task or tasks. An example of this situation can be seen in Figure 3.5. There, the $Nfib$ task is overwritten with an $Add$ task. This latter processes the results returned by the two new $Nfib$ tasks created later in the code.

In order to overwrite the $Function$ pointer of the current TS entry, the CPU only has to make a call to $OVERWRITE$. This operation changes the value of the $Function$ pointer of the TS entry pointed to by the TSP, overwriting it with the value given in the corresponding argument.

**Overwriting the Arguments Pointer**

While performing load balancing, specifically when task stealing is taking place, SLAM’s runtime system overwrites the $Arguments$ pointer in the TS entry where the stolen task is pushed. This overwriting is done by making a call to $OVERWRITE_ARGS$.

### 4.2.4 Task Stack Cache

The manipulation of the Task Stack is central to SLAM, so that aiming to produce a highly efficient implementation, custom hardware support has been provided. Having in mind that the TS must be of unlimited size to allow the solution of arbitrarily large problems it was decided to implement a TS cache hierarchy. Since most of the accesses are either pushes, pops, reads or writes at the top of the TS, the access pattern is very simple and predictable. This fact greatly simplified the implementation.

The first TS cache level, which is attached to each CPU, is implemented using a circular buffer. A pair of registers called $TS_{top}$ and $TS_{bottom}$ are used in order to keep reference to the top and the bottom of the buffer respectively at any time. When the L1 TS cache runs out of space, a block of the oldest entries is copied to the next level of the hierarchy making space for storing more tasks. On the other hand, when previously evicted entries are needed, they are brought from their location in the TS cache hierarchy. In both cases, when TS entries are copied to or retrieved from the next level of memory, the pointers stored in $TS_{top}$ and $TS_{bottom}$ are updated appropriately.

This mechanism allows a relatively small number of entries in the first level of the TS cache, enabling in this way very fast access to it. The TS cache hierarchy needs dedicated hardware only in its Level 1 (L1). The subsequent levels are kept
CHAPTER 4. SLAM IMPLEMENTATION

in the subsequent normal cache levels and in main memory.

Having a TS L1 cache allows very fast access to the TS. This fast access makes the cost of pushing and popping tasks very low. The low cost of enqueuing and dequeuing tasks allows SLAM to exploit very fine grain parallelism.

Task Stack Cache Miss

One of the issues to solve with a TS cache hierarchy is the handling of TS cache misses. These misses arise when at some point in the execution of a parallel computation, a remote CPU notifies the completion of a previously stolen task that corresponds to an entry that is not present in the L1 TS cache. In this case the absent TS entry has to have its Function pointer overwritten so that it is changed from Steal to Null. Local accesses to the cache will always occur at the top of the stack resulting in only the occasional miss. However, the above action following the completion of a remote task may occur anywhere in the stack. The worst case scenario is that such accesses could cause thrashing between local and remote entries.

A much simpler approach is just to access the information corresponding to the absent TS entry directly where it resides in the TS hierarchy. In this way there is only one access to the next level of the TS cache.

In the proposed implementation, when the value of a previously stolen task arrives at a victim CPU, but the corresponding TS entry is not present in the L1 TS cache, the TS cache miss is handled just as a normal cache miss. In this way, the Function pointer of the absent TS entry (wherever it is located in memory) is changed to Null and then the CPU continues with its previous work.

4.2.5 Task Stack Cache Hardware

The TS L1 cache implementation as already mentioned is a circular buffer built with Static Ram, Figure 4.1 shows how this buffer is structured. Each cell in this buffer has a 28-bit wide tag for identifying the address of the TS entry. Additionally there are 3 4-byte width fields for storing the pointers comprising a SLAM task descriptor. The total width of each entry is 124 bits, 28 for the tag and 96 for the task descriptor pointers. In this work a buffer of 64 entries is used for the evaluation of the system. This implies a cost of 768 bytes for storing data and 224 bytes for tags making a total of 992 bytes of Static Ram.
Figure 4.2 shows the TS L1 cache organization as well as how it is interfaced to the CPU and the immediately lower memory level. Having a 28-bit tag working with 32-bit implies that the remaining 4 bits of the address are used for indexing the 12 bytes corresponding to the task descriptor pointers. The registers dedicated to keep the TS$_{top}$ and TS$_{bottom}$ pointers are also shown.

### 4.3 Buffer of Stealable Tasks

It has been explained that SLAM executes Stealable and non-Stealable tasks. The way to distinguish between these two types of tasks is given in the source code. A flag indicates whether a task is Stealable or not when it is created. At hardware level this difference is made effective using a special buffer for storing the Stealable tasks, the Buffer of Stealable Tasks (BST).

Similarly to the L1 TS cache, the BST is implemented using a circular buffer that stores every Stealable task generated by the CPU (as long as there is available space). There are two registers whose function is storing the pointers that determine the available space in the buffer. The BST$_{bottom}$ and BST$_{top}$ pointers are the ones that point to the bottom and to the top of the BST respectively. These pointers are updated appropriately each time a new task is pushed onto or popped from the BST. The Stealable tasks are popped from the top of the BST when these are executed by the owner CPU. On the other hand, when a task is stolen, it is taken from the bottom of the BTS. In this way the BST acts as a double ended queue.
4.3.1 Buffer of Stealable Tasks Operation

Pushing a task

As has been shown in Chapter 3, not all the tasks generated by the CPU are considered as Stealable. Only those which fall in this category are pushed on the BST. This means that some task descriptors corresponding to Stealable tasks can be stored in both, the TS and the BST at certain points in the execution of a program.

Once a task is identified by the CPU as Stealable, first the BST_top pointer is incremented by 1 to point to the next entry in the buffer, and then the values of the task descriptor, as well as its corresponding address in memory, are written in the BST. This happens as long as there are empty entries in the buffer. This is the only case in which the buffer is updated with new entries. Once the buffer has reached its storage capacity, no more tasks are stored until there is some
available space.

With the implementation presented in this work, every time a task is pushed on the BST, that same task is also pushed simultaneously on the TS. However, it does not necessarily happen in the opposite way. Once the BST is full, the tasks generated by the CPU that would be considered as Stealable are only pushed in the TS. It must be noticed that only those tasks stored in the BST can be stolen by a remote CPU.

**Popping a task**

There are two cases in which a task can be popped from the buffer. The first one is the common case in which a task resident in the buffer was not stolen and its corresponding TS entry is reached by the CPU. In this case, when the CPU jumps to execute this task, simultaneously the corresponding entry in the BST (located at the top) is invalidated to avoid it from being stolen by another CPU. This invalidation occurs simply by decrementing by 1 the value of the $BST_{\text{top}}$ pointer.

The second case occurs when a task is taken from the BST in order to serve a remote request. In this case the task located at the bottom of the BST is sent to the requesting CPU and executed by it. After popping the corresponding task, this is invalidated by incrementing value stored in the $BST_{\text{bottom}}$ pointer, making it to point to the next entry of the BST.

Since the tasks that are stolen are taken from the bottom of the BST, each stolen task is the oldest one stored in the buffer at any time. This favours programs that use divide and conquer algorithms as older tasks tend to have a coarser granularity.

### 4.3.2 Buffer of Stealable Tasks Hardware

Each entry in the BST has the same structure as an entry in the TS. This is a pointer to the function to be executed, one to the arguments of that function and one to the address where the returned value must be stored. Additionally each entry in the BST also stores the memory address that corresponds to the TS entry that it contains. This address is used in order to keep record of the TS entry where the task was placed originally. Having this information, a victim CPU would know what is the TS entry whose *Function Pointer* has to be written.
with a Null task when notified of the completion of a stolen task.

The BST allows a remote request to be served very efficiently. It ensures that Stealable tasks are always in the immediate scope of the CPU, even if those tasks are not resident in the L1 TS cache due to having previously been evicted.

Figure 4.3 shows the hardware structure of the BST. It is assumed that the BST is built with static Ram. Each cell comprises 12 bytes for storing a task descriptor and 28 bits for storing the corresponding TS entry memory address. The BST used in this work has 32 entries requiring 384 bytes for storing task descriptors and 112 bytes for storing the memory addresses, making a total of 496 bytes.

![Figure 4.3: Hardware structure of the Buffer of Stealable Tasks.](image)

### 4.3.3 Communication Control

As mentioned in 3.1.4 SLAM was designed originally with the idea of using Active Messages [76] as the communication mechanism for achieving load balancing. In the implementation proposed in this thesis, it was decided to separate completely the communication from the computation. For this effect, a mechanism separated from the CPU performs all the SLAM related communication operations.

In Active Messages, each message contains the address of a user level handler whose function is to extract the message from the network and insert it into the ongoing computation. In this work the user level handler is implemented in hardware. When a message arrives at a processing node, it is extracted by a controller that performs the corresponding operations. This controller is called the Communication Control (CC).

The CC synchronizes all the communication tasks related to SLAM. It is a
finite state machine that handles the messages generated by SLAM's load balancing system. It deals with the requests for a task going to and coming from remote CPUs, as well as with the generation and handling of notifications related to the completion of stolen tasks.

Each time the CC gets a request from a remote CPU, it checks whether there is a task available for being exported in the BST. In the case that a task can be exported, the CC pops it from the BST and sends it to the thief CPU. In the case that there is no available task for being exported, the CC simply sends a refusal to the thief.

With this mechanism the victim CPU is not interrupted when a remote request arrives as long as it does not try to access its TS while the request is being served by the CC. If this happens, the CPU gets stalled until the request has been served, after which it can proceed safely.

4.3.4 SLAM Network

In order to establish communication among the CPUs in the system, the existence of a NoC (Network on Chip) is assumed. In the proposed implementation this NoC is used to carry messages related to dynamic load balancing. These messages can be:

- A request for a task. This is generated by a thief CPU each time it executes a *Steal* task.
- An answer to a remote request. This answer can be a task or a refusal.
- A notification of a stolen task completion. This is generated when a *Return* task is executed by a thief CPU.

4.3.5 SLAM Hardware organization

Figure 4.4 shows the hardware organization of all the components of SLAM. The registers dedicated to keep the $TS_{top}$, $TS_{bottom}$, $BST_{top}$ and $BST_{bottom}$ pointers are shown in the shaded area, together with the *Task Stack*, the *Buffer of Stealable Tasks* and the *Communication Control*. The *Task Stack Pointer* register is located inside the CPU.
4.4 Instruction Set Architecture Extension

Extensions are made to the Instruction Set Architecture (ISA) of the CPUs to handle the operations performed in the TS. These operations are reads or writes performed on TS entries. The instructions added to the ISA are shown in Table 4.1. It also shows the number of cycles they take to be executed.

All these instructions act on the TS entry pointed by the current TSP at each time. This means that all the TS entries impacted by these instructions are cached in the L1 of the TS cache hierarchy.

4.5 Stealing Process

4.5.1 Initiating a Steal

The stealing process starts when a CPU executes a Steal task becoming a thief. When executing this task the thief transfers the control to its Communication Control (CC). Then the CC proceeds to select a victim (the CPU from which
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG_ADDR</td>
<td>Gets the memory address where the arguments of the function resident in the TS entry pointed by the TSP are stored.</td>
<td>2</td>
</tr>
<tr>
<td>RET_ADDR</td>
<td>Gets the memory address where to store the result produced by the function resident in the TS entry pointed by the TSP.</td>
<td>2</td>
</tr>
<tr>
<td>OVERWRITE</td>
<td>Overwrites the Function pointer of the TS entry pointed by the TSP with a different value.</td>
<td>2</td>
</tr>
<tr>
<td>PUSH</td>
<td>Pushes a new entry in the TS.</td>
<td>5</td>
</tr>
<tr>
<td>NEXT_TASK</td>
<td>Jumps to the address pointed by the Function pointer resident in the TS entry pointed by the TSP.</td>
<td>2</td>
</tr>
<tr>
<td>DEC_TSP</td>
<td>Decrements the TSP making it to point to the previous TS entry.</td>
<td>1</td>
</tr>
<tr>
<td>INC_TSP</td>
<td>Increments the TSP making it to point to the next TS entry.</td>
<td>1</td>
</tr>
<tr>
<td>PUSHREM</td>
<td>Pushes a new entry in the TS of a remote CPU.</td>
<td>6</td>
</tr>
<tr>
<td>OWRTREM</td>
<td>Overwrites the TS entry pointed by the TSP in a remote CPU.</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 4.1: ISA extensions added by SLAM. The PUSHREM and OWRTREM operations only take into account the cycles spend by the CC to form the appropriate information packet. The time that the mentioned packet travels through the network is not included.
a task is going to be stolen). This selection is done randomly. Next it builds an information packet containing the number of the thief and the appropriate code for a Steal Request. Finally this packet is sent to the victim through the SLAM network. The thief is stalled until it gets an answer from the victim. Figure 4.5 shows a flowchart with the sequence of actions performed by the CC when initiating a steal. The initiation of a steal requires 5 cycles to be performed.

![Flowchart](image)

Figure 4.5: Actions performed by the CC when initiating the steal of a task.

### 4.5.2 Serving a Steal Request

When the CC of the victim receives the request from the thief, it checks whether there is an available task for being exported in the BST. There are two possibilities in this case:

1. There is a task available in the BST. In this case the victim sends this task to the thief.

2. The BST is empty. In this case the victim sends a refusal indicating that it does not have any task for being exported.

**Sending a task**

When sending a task to a thief, the victim’s CC sends an answer containing the stolen task and information that allows the victim CPU to know where the stolen task was residing in the TS before being stolen.

The actions performed by the CC are listed below:
1. Overwrite the *Function pointer* residing in the TS entry of the task being stolen with a pointer to a *Steal* task.

2. Form an information packet containing the number of the victim, a pointer to a *Return* task and a pointer to the address of the TS entry where the task being stolen was originally placed. As mentioned earlier, this information allows the victim CPU to know where to set the pointer to a *Null* task when it is notified about the completion of a stolen task.

3. Send the first information packet through the network.

4. Form an information packet containing the task descriptor residing in the BST entry pointed by the *BST bottom* pointer.

5. Send the second information packet through the network.

6. Increment by one the *BST bottom* pointer.

The operations performed when sending a task require a total of 9 cycles if the TS entry that contains the task being stolen is resident in the first level of the TS cache hierarchy. From these 9 cycles 2 are for retrieving the address of the TS entry of the task being stolen, 2 are for writing the pointer to a *Steal* task in the corresponding TS entry, 1 is for building the first packet of information for the thief, 1 is for sending this packet, 2 are for reading the data of the TS entry from the BST and form the second packet of information for the thief, 1 is for sending this packet through the network.

If the TS entry that contains the task being stolen is not resident in the first level of the TS cache hierarchy and, assuming that this entry is located in the second level of the above hierarchy, then writing the pointer to a *Steal* task in the corresponding TS entry would take 12 cycles. This corresponds to the latency of the level 2 cache. In this way, for this particular case, the total number of cycles for sending a task to a thief would be 19.

**Sending a refusal**

When sending a refusal to a thief, the victim’s CC sends an answer containing a code that indicates the thief that its request was not successful.
In this case the CC forms an information packet containing a *Refusal* code and sends it to the thief through the network. The number of cycles required when sending a refusal is 5.

The sequence of actions performed by the CC when serving a request from a thief are shown in the flowchart in Figure 4.6.

**Figure 4.6: Actions performed by the CC when serving a request for a task coming from a thief CPU.**

### 4.5.3 Receiving Victim’s Answer

When the CC of the thief gets the answer from the victim, there are two possible scenarios:

1. The answer contains a task indicating a successful steal.
2. The answer is a refusal indicating an unsuccessful steal.
Receiving a task

When the CC of the thief gets the first part of the answer from the victim, it simply pushes the content of received packet (the victim’s CPU number, the TS entry memory address and the pointer to a Return task) onto the TS; then it proceeds to wait for the second part of the answer. When the second part is received, the content of the packet (the task descriptor of the stolen task) is pushed onto the TS. Finally the CC notifies the thief giving it control of the processing node.

The number of cycles for performing the operations related to receiving a task is 9. 1 is for checking whether the victim’s answer contains a task or a refusal, 2 for pushing the content of the first packet of information on the TS of the thief, 2 for waiting the arrival of the second packet\(^1\), 2 for pushing the content of the second packet on the TS and 2 for notifying the CPU that the stolen task is ready for execution.

When the thief is notified, it proceeds to copy the arguments of the stolen task from the arguments array of the victim to its local one. Once the arguments have been copied to the local arguments array, the thief can proceed to execute the recently stolen task.

Copying the arguments of the stolen task from the victim’s arguments array to the one belonging to the thief is a safe process. This is because even if the victim reaches the TS entry where the stolen task was placed, it would execute the Steal task that replaced the stolen task (see Section 3.2.1 in Chapter 3). This task does not use any arguments, meaning that the victim would not try access the arguments corresponding to the TS entry of the stolen task. In this way the thief CPU can copy the arguments of the stolen task without any risk of getting out of date values.

The copying of the arguments allows the thief to read and write arguments for the subsequently created tasks always accessing its local arguments array.

Receiving a refusal

When the CC of the thief gets a refusal from the victim it notifies the CPU, which in turn, initiates the stealing process again. The CC requires 3 cycles for notifying the CPU when it gets a refusal.

\(^1\)It is assumed that there is no contention on the network.
The actions performed by the CC when receiving the answer corresponding to a previous request are illustrated in the flowchart shown in Figure 4.7.

![Flowchart](image)

Figure 4.7: Actions performed by the CC when getting the answer corresponding to a request made to a victim CPU.

### 4.5.4 Notifying the Completion of a Stolen Task

The **thief** notifies the completion to the **victim**

The *Return* task is responsible for notifying the completion of a previously stolen task to its original owner. This task is executed by the thief immediately after the stolen task has been evaluated.

When a *Return* task is executed, the CPU transfers control to the CC and then stalls. After this, the CC gets the CPU number of the corresponding victim.
Next it forms an information packet containing a pointer to a Null task and the address of the TS entry where the stolen task was originally placed before the steal (the number of the victim, as well as the TS entry address are stored in the current TS entry of the thief). Then this packet is sent to the victim through the network. Finally the CPU is notified and it proceeds with the execution of the next task stored in its TS.

The CC requires 7 cycles for executing the operations related to the notification of the completion of a stolen task. 2 for reading the number of the victim, 2 for reading the address of the TS entry where the stolen task was originally placed, 1 for sending the packet containing the above address and the pointer to a Null task and 2 for notifying the CPU to proceed with the execution of the next task.

The actions performed by the CC when notifying the completion of a stolen task to a victim are illustrated in the flowchart shown in Figure 4.8.

![Flowchart: Actions performed by the CC when notifying a victim CPU about the completion of a stolen task.](image)

Figure 4.8: Actions performed by the CC when notifying a victim CPU about the completion of a stolen task.

**The victim receives the notification from the thief**

When the CC of the victim CPU gets the notification about the completion of a previously stolen task, it stalls the CPU. Next it writes the pointer to a Null task in the corresponding cache stack entry. Finally the CPU is signaled to continue with its previous work.

It should be noticed that the CC checks if the TS entry from which the remotely completed task was stolen is resident in the L1 TS cache. If that is
the case, then the CC can write directly to the corresponding cached TS entry. Otherwise it has to be written in the place where it is located in the memory hierarchy.

If the TS entry of the task that has been notified as completed is resident in the the first level of the TS cache hierarchy, then the CC needs 6 cycles for executing the operations associated with the receipt of the above notification. 2 for stalling the CPU, 2 for storing the pointer to a Null task in the corresponding TS entry and 2 for giving control of the processing node to the CPU.

If the corresponding TS entry is located in the second level of the S cache hierarchy, then the storing of the pointer to a Null task would take 12 cycles, corresponding to the latency of the level 2 cache. In this way, for this particular case, the CC would need a total of 16 cycles for executing all the associated operations.

The actions performed by the CC when getting the notification about the completion of a stolen task from a thief are illustrated in the flowchart shown in Figure 4.9.

![Figure 4.9: Actions performed by the CC when getting the notification about the completion of a stolen task from a thief CPU.](image)

4.5.5 Messages Involved in Task Stealing

As it has been explained, SLAM performs load balancing dynamically using work stealing. The stealing process requires to send and receive information through the SLAM network. The messages involved in SLAM’s dynamic load balancing are shown in Table 4.2. This table also shows the information that is contained in each message, as well the corresponding sender and receiver.
CHAPTER 4. SLAM IMPLEMENTATION

<table>
<thead>
<tr>
<th>Message</th>
<th>Content</th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEAL_TASK</td>
<td>Thief’s CPU number</td>
<td>Thief</td>
<td>Victim</td>
</tr>
<tr>
<td>REFUSE_STEAL</td>
<td>Victim’s CPU number</td>
<td>Victim</td>
<td>Thief</td>
</tr>
<tr>
<td>TASK_P1</td>
<td>Victim’s CPU number, TS entry memory address, Pointer to a Return task</td>
<td>Victim</td>
<td>Thief</td>
</tr>
<tr>
<td>TASK_P2</td>
<td>Descriptor of the stolen task</td>
<td>Victim</td>
<td>Thief</td>
</tr>
<tr>
<td>TASK_FINISHED</td>
<td>TS entry memory address, Pointer to a Null task</td>
<td>Thief</td>
<td>Victim</td>
</tr>
</tbody>
</table>

Table 4.2: Messages involved in SLAM’s Dynamic Load Balancing.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
<th>CPU Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiate Steal</td>
<td>5</td>
<td>Thief</td>
</tr>
<tr>
<td>Send a Task</td>
<td>9-19</td>
<td>Victim</td>
</tr>
<tr>
<td>Send a Refusal</td>
<td>5</td>
<td>Victim</td>
</tr>
<tr>
<td>Receive a Task</td>
<td>9</td>
<td>Thief</td>
</tr>
<tr>
<td>Receive a Refusal</td>
<td>3</td>
<td>Thief</td>
</tr>
<tr>
<td>Notify Completion</td>
<td>7</td>
<td>Thief</td>
</tr>
<tr>
<td>Receive Notification</td>
<td>6-16</td>
<td>Victim</td>
</tr>
</tbody>
</table>

Table 4.3: Timing assumed for Communication Control operations.

4.5.6 Timing of Communication Control Operations

Table 4.3 shows the assumed execution times expressed in CPU cycles for each of the CC’s operations. It also shows the role that the CPU plays in each operation.

In the case of the Receive a Task operation (see Figure 4.7), the assumed execution time comprises pushing the first part of the answer on the TS of the thief, waiting for the second part and finally pushing it. It is assumed that the involved network packets traveled without facing any contention in the interconnect.

For the CC operations Send a Task and Receive Notification there is a variation in the execution time. This variation is due to it is being considered whether the TS entry is located in the L1 Task Stack Cache or, if on the other hand, it is located in the L2 cache. In the case that the TS entry is resident in the main memory, then the corresponding memory access time should be considered (see Table 5.3 in Chapter 5 for information related to the assumed memory access times of the simulated systems).
4.5.7 Race Conditions

SLAM CC works as an interrupt based communications mechanism. This makes synchronization very simple since there is no need to deal with race conditions. As explained before, every time a remote request arrives at a victim CPU, the CC deals with it locally. In the meantime the victim does not need to be aware of it, except if it tries to access the TS when a request is being served. In such a case the victim is stalled until the request has been served and, in consequence, the TS has also been released. For this reason, in SLAM, CPUs do not compete for a task at any moment in the execution. If it happened that the last task to be executed was sent to a thief, then the victim would execute the Steal task that was overwritten in the TS entry where the stolen task was placed.

4.5.8 Potential Deadlock Avoidance

While waiting for the victim’s answer, it is possible to have the situation in which on one side a thief sends a steal request to a victim. On the other side the previous victim has already sent a steal request to the previous thief. In this case, there would be a time in which both CPUs would be waiting for the corresponding answers. If there is no mechanism to deal with the issued steal requests, this situation would lead to a deadlock scenario since both CPUs would be waiting for each other’s answer indefinitely.

The CC handles this kind of situations as follows:

When a CPU is waiting for an answer to a recently made steal request and another steal request from a remote CPU (which is also waiting for its answer) arrives, the CC of the first waiting CPU simply answers to the remote request with a refusal. In this way the remote CPU getting the refusal will try a different victim in its search for work.

If in the above situation happens that the first waiting CPU gets a steal request issued by its previously chosen victim, then once the steal requests get to the CC’s of the corresponding CPUs, these will answer with a refusal. When these refusals get their destination, the involved CPUs will search for work trying with different victims, avoiding in this way a deadlock.
4.5.9 Handling Multiple Messages

Another situation that is possible is that a CC that is already performing an operation can get a message requesting it to do some other operation. In this case the CC needs a mechanism to store the new message, so that when it finishes with the current operation, it can proceed to perform the next one.

This can be handled using a buffer for the incoming messages attached to each CC. This can be implemented with a circular buffer acting like a FIFO (First In First Out) queue. With this mechanism the arrival ordering of the messages is preserved and multiple incoming messages can be served. If the mentioned buffer overflows due to a high number of messages, a software mechanism backs up the overflowed entries so that these can be processed at the proper time. A similar mechanism is proposed in [22].

4.6 Example of Operation

Now that the SLAM implementation carried out in this work has been explained, the example used in Chapter 3 Section 3.2.1 is revisited showing the way in which all the components in the system work together.

Figure 4.10 shows the execution of $Nfib(5)$ in a dual core machine. The TSs and arguments arrays corresponding to CPU0 and CPU1 are shown. For simplicity, only the BST corresponding to CPU0 is illustrated.

Figure 4.10(a) shows how $Nfib(5)$ has been expanded by CPU0, generating two Add and three Nfib calls in TS0. The arguments corresponding to each of the generated calls are stored in A0 (A0 and A1 are the arguments arrays corresponding to CPU0 and CPU1 respectively). The point to notice here is that only the Nfib calls have been pushed into BST0 whilst the Add calls reside only in TS0. As explained before, this is because only the first ones can generate further parallelism in case of being stolen by a remote CPU. At this point CPU0 is about to execute $Nfib(1)$ (pointed by TSP0) and CPU1 is about to execute the Steal task resident in TS1.

It should be noted that it is assumed that CPU0 has not yet started the execution of $Nfib(1)$ (CPU0 has not jumped to the memory address referenced by the Function pointer of the TS0 entry pointed by TSP0). In consequence, the corresponding BST0 entry has not been invalidated at this point.

In Figure 4.10(b) CPU0 has finished the execution of $Nfib(1)$ and placed the
produced result in the proper memory location (the results are highlighted in A0). It can be seen that \( BST_{\text{top}} \) was decremented by ‘1’ and the corresponding BST0 entry was invalidated (the invalidated BST0 entries are highlighted). Also CPU1 has stolen the call to \( NFib(4) \) which now is resident in TS1. This call was taken from the bottom of BST0 and sent to CPU1. \( BST_{\text{bottom}} \) was incremented by ‘1’ invalidating the BST0 entry of the stolen task. At this point, CPU1 has already copied the arguments of the recently stolen task from A0 to A1. This can be seen at entry number ‘2’ in A1. In this way, all the arguments corresponding to functions created by CPU1 will be created in A1.

Figure 4.10(c) shows the situation in which CPU0 has finished the execution of \( NFib(2) \). BST0\text{top} has been decremented by ‘1’ and the proper BST0 entry has been invalidated leaving BST0 empty. On the other hand, CPU1 has finished the execution of \( NFib(4) \) and placed the produced result in entry number ‘0’ at A0. It is also about to execute the Return task resident in TS1. When CPU1 executes this task, it notifies CPU0 about the completion of the stolen task. This is done by overwriting the Function pointer of the entry from which the task was stolen originally. The mentioned pointer is overwritten with a pointer to a Null task. The memory address of the entry in which the stolen task was originally placed is stored in TS1. It resides in the Return pointer of the TS1 entry pointed by TSP1.

At this point, the result of the stolen task has been placed in the proper memory address, however the CPU0 has not been notified about the completion of the stolen task. If CPU0 reached the entry number ‘1’ in TS0 before this notification occurs, it would execute a Steal task, not being allowed to pass that TS0 entry until the mentioned notification happens. In this way the Steal task constitutes a synchronization point; it does not allow CPU0 to execute a function whose arguments are not available yet.

In Figure 4.10(d) CPU1 has executed the Return task which sent a notification about the completion of a stolen task to CPU0. When the CC in CPU0 got the notification it overwrote the Function pointer of the TS0 entry from which the task was stolen. It can be seen in entry number ‘1’ at TS0 where the Steal task was replaced by a Null task.

CPU0 is executing the Add call located in entry number ‘2’ at A0. When it executes the Null task located in the following TS0 entry, it will simply jump to execute the function stored in entry number ‘0’ at TS0.
Figure 4.10: \textit{Nfib(5)} computed in parallel in a dual core machine using the proposed SLAM implementation.
4.7 Full System Implementation

In order to evaluate the proposed SLAM implementation, simulated tiled cache coherent Chip Multiprocessors (CMPs) ranging from 4 to 64 cores were used. The tiles are interconnected using a 2-Dimension Mesh configuration. Figure 4.11 shows how the system is organized inside the CMP. Each tile contains one core, Level 1 data and instructions caches, a Level 2 unified cache, a directory for cache coherency purposes, a router and the hardware used for implementing SLAM. Inside the SLAM hardware section reside the Communication Control, the Level 1 Task Stack cache and the Buffer of Stealable Tasks. The processor interconnect is used to carry messages related to SLAM’s runtime system.

At this point it can be noticed that the proposed SLAM implementation has some similarities with Carbon and ADM. In SLAM all the operations performed on the TS make use of custom hardware, the same as Carbon does with the queues of its Global Task Unit (GTU). However SLAM’s runtime system provides a higher flexibility than Carbon. This is because in SLAM the custom hardware can be used by the software runtime system for implementing different scheduling policies. Carbon on the other hand, has the scheduling policies completely hardwired in the GTU.

On the other hand SLAM is similar to ADM in that both systems implement their dynamic scheduling systems in software, making use of custom hardware for increasing efficiency. Also both systems make use of the interconnection network for sending/receiving the messages related to load balancing. However in SLAM the custom hardware is relatively more complex than the proposed in ADM. Also since the TSs are central for SLAM’s operation, its runtime system cannot be as flexible as the ones for ADM, for which the central element is the low-overhead inter-core messaging functionality. On the other hand, SLAM’s runtime system is simpler than ADM’s due to the fact that several operations are performed by fixed-function hardware.

4.8 Summary

This chapter has provided a detailed description of the SLAM implementation presented in this work. It started with a description of the TS as well as the way it operates. This included the TS cache implementation used in the system
evaluation. Next the BST was introduced with the corresponding description of its operation and the hardware used for its modeling. The ISA additions made were also covered, showing the number of CPU cycles each new instruction takes when executed.

Regarding dynamic load balancing, the CC was described explaining the different stages in the process of stealing a task from a remote CPU. This included both, the actions performed by the CC residing in the thief CPU, as well as those executed by one residing on the victim. The description of the network messages involved in the task stealing process were also covered.

This chapter also provided an explanation about how race conditions are avoided using the CCs in each CPU. This is basically because all the operations performed on the TS of each CPU are carried out locally by their corresponding CC. This eliminates the possibility of competition for resources residing in the TS of any CPU.

In order to show the way in which the whole system interacts, an example was provided. This example showed how each element of the proposed implementation work when executing a parallel application using SLAM.

Finally, a description of the full system implementation was presented showing the way in which the overall system is organized.

The next chapter presents the evaluation of the system. It introduces the simulation tools used for building the multiprocessor system environment, as well as the custom hardware modeling. It also presents a performance comparison of a system with SLAM implemented versus a base line system running Cilk [29]. Finally it provides a discussion of the results obtained from the experiments performed during the evaluation of SLAM.
Chapter 5

SLAM Evaluation

5.1 Introduction

The previous chapter provided a description of the implementation of SLAM developed in this work. This included the individual hardware elements used for implementing the Task Stack and the Buffer of Stealable Tasks, as well as a description of the Communication Control attached to each processing node. It also described the role of the network used to interchange messages related to SLAM’s dynamic load balancing.

This chapter provides a description of the experimental work carried out in this thesis. This work involved a performance comparison between SLAM and Cilk [29] which was performed using a set of divide-and-conquer applications. As mentioned in the previous chapter, this comparison was made using simulated cache coherent tiled CMPs interconnected using a 2-Dimension mesh network.

The performance comparison for the divide-and-conquer applications is presented in two parts. The first part examines the performance in terms of the execution time of the applications tested in both systems. The second part examines the efficiency of the dynamic load balancing systems in terms of stealing time and steal granularity. The first one corresponds to the time each CPU spent stealing work from other CPUs when executing each application. The second one refers to the time each CPU spends doing useful computation when it completes a successful steal before it tries to steal again.

Additionally a couple of loop-level parallel applications were also evaluated. This evaluation shows the performance comparison between SLAM and the equivalent version of each benchmark coded using the Pthreads [15] library.
The description of the experimental work starts by providing the details of the way in which the evaluated systems were simulated. Then, the simulation tools used for the experiments are presented. The chapter continues with a description of the benchmarks employed. Next, the settings of the simulated systems are covered. Finally, the results of the experiments are presented and discussed.

5.2 Simulation Methodology

The experiments for evaluating SLAM’s performance were run using simulated cache coherent CMPs with SLAM hardware support. A set of applications written using the SLAM’s programming interface discussed in Chapter 3 Section 3.3.1 were executed on the mentioned CMPs.

Similarly, the experiments for evaluating Cilk’s performance were also run using simulated cache coherent CMPs. Since Cilk is completely implemented in software, it is assumed that the CMPs executing Cilk applications did not have any additional custom hardware implemented.

The CMPs used in the experiments were modeled using Simics [54], a full system functional simulator. The memory hierarchy was modeled using GEMS [55], a framework built based on Simics that provides cycle accurate timing simulation for multiprocessor memory systems. The processor interconnect was modeled using GARNET [64], an interconnection network model based on GEMS.

The custom hardware used for implementing SLAM was modeled inside a separate Simics module called SLAM module. The L1 Task Stack (TS) Cache, the Buffer of Stealable Tasks, and the Communication Control (CC) were handled using calls to the SLAM module. These calls were implemented in the form of magic instructions. These kind of instructions are a functionality provided by Simics. Using this functionality it is possible to model the behavior and timing of additional hardware components added to the simulation. Every call to the SLAM module performs an operation that acts either on the TSs or the CCs of the CPUs simulated in the system.

The complete system simulated gets information from 4 different sources: Simics, GEMS, GARNET and the SLAM module. Table 5.1 shows the information that each of these components provides to the overall system.
<table>
<thead>
<tr>
<th>Component</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simics</td>
<td>Full system functional simulation.</td>
</tr>
<tr>
<td>GEMS</td>
<td>Cycle accurate memory system simulation.</td>
</tr>
<tr>
<td>GARNET</td>
<td>Cycle accurate interconnection network simulation.</td>
</tr>
<tr>
<td>SLAM module</td>
<td>Cycle accurate SLAM hardware simulation.</td>
</tr>
</tbody>
</table>

Table 5.1: Information provided by each simulation tool to the overall system simulation.

5.3 Simulation Tools

5.3.1 Simics

Simics [54] is a full system functional simulator, it has a level of functional accuracy sufficient for running commercial workloads. At the same time its timing accuracy allows it to interact with detailed hardware models. Furthermore, Simics’ level of abstraction delivers a reasonable performance leading to tolerable simulation times.

Simics allows the running of applications on top of an unmodified operating system (OS). Having OS support provides access to thread scheduling and memory management functionalities. This fact allows the execution of multithreaded workloads in a simulated multiprocessor machine. On a multiprocessor system, Simics switches among processors in a round robin fashion during the execution of instructions.

Simics’ functionality can be extended to produce more detailed simulations, as well as to model elements not considered in the original system. This extension can be made using modules. A Simics module is a piece of software with which it is possible to interact with the simulator using the Simics API. The functions provided in this API allow the interchange of information between Simics and a module, providing the possibility of changing the default behavior of a simulation. This feature allows the modeling of elements not considered at a functional level, for example the memory hierarchy or any other additional hardware. In this work, Simics is used as the main simulation platform for providing the functional simulation of the whole system.

As mentioned earlier, SLAM infrastructure is built using a separate Simics module. The Task Stacks, Buffers of Stealable Tasks and Communication Controls, as well as the corresponding timing are modelled inside this module.
CHAPTER 5. SLAM EVALUATION

5.3.2 GEMS

Being a functional simulator, Simics does not model any hardware at micro-architectural level by default. In order to model more detailed systems like a multiprocessor memory system, Simics must be complemented using customized modules capable of simulating in detail such a system.

The memory system of the proposed SLAM implementation is modeled using the Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset [55]. It decouples simulation functionality and timing using the timing-first simulation approach [57]. GEMS is implemented in the form of Simics modules. It provides timing information that indicate Simics when to execute an instruction. However, the result of the execution of that instruction depends only on Simics. In this way GEMS provides the timing of the memory system, whilst Simics provides the correctness of the execution.

It is worth mentioning that although timing and functionality are handled separately, the functional simulation is affected by the timing simulation and vice versa. This fact allows the system to capture timing dependent effects providing a more detailed simulation as a whole.

Ruby

The core of GEMS is Ruby, which is a memory system simulator. It uses the GEMS Simics driver to approximate a simple in-order processor without pipeline stalls. Simics passes all the memory accesses to Ruby (loads, stores and instruction fetches). Ruby checks if a memory access is a hit or a miss in the L1 cache. In the case of a hit, the requesting processor is only stalled during the L1 cache access time, after which it continues executing the following instructions. On a miss, the requesting processor is stalled while Ruby simulates the cache miss. Each processor in the system can only have one miss outstanding. Issues like contention and other timing effects determine when each request is completed. In this way, the functional simulation performed by Simics becomes timing-dependent.

Ruby models caches, cache controllers, memory controllers and banks of main memory. Ruby uses a queue-driven event model for simulating timing. The different components establish communication using message buffers of different latencies and bandwidths. Each component at the receiving end of a buffer is scheduled to wake up when there is a new message available to be read from the corresponding buffer. The simulation continues calling a wakeup method for the
next scheduled event resident in the event queue.

The Specification Language for Implementing Cache Coherence (SLICC) [55], a domain specific language included in GEMS, is designed for the creation of different cache coherence protocols and memory hierarchies. Individual controller state machines like cache controllers and directory controllers can be defined using SLICC. Each controller includes protocol-independent components such as cache memories and directories. A memory system can be defined using a SLICC specification. Taking that specification, the SLICC compiler produces C++ code. At the end, this code is linked with the protocol independent portions of the Ruby memory system simulator. In this way, the use of SLICC simplifies and accelerates the development of memory systems aimed to work in Ruby.

5.3.3 Garnet

In this work, CMPs with up to 64 cores were evaluated. For CMPs of this size, the processor interconnect plays a very significant role in the system evaluation. In consequence, the interaction between the memory hierarchy and the interconnection network has to be taken into account when evaluating large CMPs. The interconnection network of the CMPs evaluated in this work is modeled using GARNET [64], a detailed on-chip network model inside a full system simulator. It models in detail the interconnection network at a microarchitecture level. GARNET is included in the GEMS toolset [55].

GARNET models a state-of-the-art on chip network that uses a modular packet-switched fabric. In this fabric, network channels are shared by multiple packet flows. In GARNET, a five-stage pipeline router with virtual channel flow control [20] is implemented. The major components of this router are the input buffers, route computation logic, virtual channel allocator, switch allocator and crossbar switch.

The communication operations of the entire memory system are done using the interconnection network. All the Level 1 and Level 2 cache controllers, as well as the memory controllers communicate with each other using GARNET’s interconnection network.

Other features provided in GARNET include point-to-point ordering, which is modeled implementing system-level ordering support in the VC and switch allocators. Also, the power consumption of the interconnection network is calculated. This is done using the Orion power models [79] implemented in the simulator.
5.4 Benchmarks

From the benchmarks evaluated, \textit{LU}, \textit{FFT}, \textit{Cholesky} and \textit{Heat} were translated to SLAM code directly from the examples provided with Cilk 5.4.6. \textit{Nfib} and \textit{N Queens} were adapted from the same examples. In the case of \textit{Nfib}, the function definition was modified to provide the desired evaluation. For \textit{N Queens}, the code was adapted in order to make the program to find all the possible solutions for a certain number of queens. \textit{Merge Sort} and \textit{Quick Sort} were written from scratch due to the simplicity of their corresponding implementations.

The serial versions are the \textit{C elisions} [29] of the corresponding Cilk versions. A \textit{C elision} is the result of eliminating Cilk keywords from a Cilk application. This result is a semantically correct C program.

The benchmarks evaluated are divided into divide-and-conquer applications and loop-based applications.


The loop-based applications are: \textit{Matmult} and \textit{Integration}.

Table 5.2 shows the input used for each benchmark in the experiments performed.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfib</td>
<td>40</td>
</tr>
<tr>
<td>N Queens Low</td>
<td>13 queens</td>
</tr>
<tr>
<td>N Queens High</td>
<td>14 queens</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>4 million random doubles array</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>5 million random doubles array</td>
</tr>
<tr>
<td>LU Low</td>
<td>1024x1024 matrix</td>
</tr>
<tr>
<td>LU High</td>
<td>2048x2048 matrix</td>
</tr>
<tr>
<td>FFT Low</td>
<td>$2^{20}$ elements vector</td>
</tr>
<tr>
<td>FFT High</td>
<td>$2^{21}$ elements vector</td>
</tr>
<tr>
<td>Cholesky</td>
<td>1000x1000 matrix with 10000 non zeros</td>
</tr>
<tr>
<td>Heat</td>
<td>1024x512 mesh</td>
</tr>
<tr>
<td>Matmult</td>
<td>512x512 matrix</td>
</tr>
<tr>
<td>Integration</td>
<td>$f(x) = x^3$ from 0 to 10 with 8,388,608 rectangular sections</td>
</tr>
</tbody>
</table>

Table 5.2: Inputs used in the system evaluation.
5.4.1 Nfib

The $Nfib$ benchmark evaluates the Nfib function for $n$. It is defined as:

\[
Nfib(n) = 1, \text{ if } n < 2
\]

\[
Nfib(n) = 1 + Nfib(n-1) + Nfib(n-2), \text{ if } n \geq 2
\]

The tasks executed in this benchmark are very fine grained, so that it is a good reference to show how SLAM behaves when exploiting very fine grained parallelism.

5.4.2 N Queens

The $N Queens$ benchmark finds all the possible solutions for the N queens problem. The objective is to place $n$ queens on an $nxn$ chessboard, in a way that no pair of queens share the same row, column or diagonal. It uses a backtracking depth-first search algorithm.

5.4.3 Merge Sort

The $Merge Sort$ benchmark orders an array of $n$ randomly generated integers using the Merge Sort algorithm. It divides recursively the array to be sorted in two subarrays until reaching the base case. The base is reached when the subarrays have one or zero elements. In the merge phase, the two already sorted subarrays are merged to form a single sorted array.

5.4.4 Quick Sort

The $Quick Sort$ benchmark orders an array of $n$ randomly generated integers using the Quick Sort algorithm. An element of the array to be sorted is chosen as a pivot. Then the array is divided in two subarrays, one containing smaller values than the pivot value, and the other one containing bigger values. This division is made recursively until the entire array is sorted.
5.4.5 LU

The LU benchmark performs the divide and conquer form of a blocked LU decomposition of a dense matrix into the product of two triangular matrices, Lower and Upper \((A = LU)\). The division phase creates submatrices until it reaches a threshold. From that point on, the LU decomposition is calculated sequentially for each submatrix. In this way, multiple calculations are performed in parallel.

5.4.6 FFT

The FFT benchmark performs the recursive Fast Fourier Transform on the \(n\) complex components of an array. The result is placed in a different array. It uses a basic Cooley-Tukey algorithm.

5.4.7 Cholesky

The Cholesky benchmark performs a Cholesky factorization on a random sparse symmetric positive definite matrix \((A = LL^T)\). The nonzero entries of the sparse matrix are stored in a quad-tree. The critical path as well as the actual work depend on the sparsity pattern of the matrix. It uses a divide and conquer algorithm, which after reaching a threshold performs the factorization sequentially.

5.4.8 Heat

The Heat benchmark performs the Heat-diffusion calculation on a \(m \times n\) mesh. It uses a Jacobi-type iteration to solve the finite-difference approximation of the partial differential equations that can model the heat diffusion problem. The 2D-mesh is partitioned into submeshes. Then each resulting submesh is divided into stripes and then each stripe is computed. Multiple stripes are computed in parallel.

5.4.9 Matmult

The Matmult benchmark calculates the multiplication of two matrices. One matrix is divided into uniform stripes. These are distributed to the available processing cores in the system to perform a parallel calculation.
The SLAM version of this benchmark is compared against a version coded using the Pthreads library implementing static scheduling. The purpose of this was to evaluate SLAM when working with loop-level parallel applications.

5.4.10 Integration

The Integration benchmark calculates the area under the curve produced by the function \( f(x) = x^3 \). This benchmark employs a finite approximation in which the area is divided into rectangular sections. A subset of these is distributed to the available processing cores in the system and the calculation is performed in parallel.

The same as Matmult, Integration is a loop-level parallel application. The SLAM version of this benchmark is compared against a version coded using the Pthreads library implementing static scheduling.

5.5 Simulation Environment

In order to evaluate the proposed SLAM implementation, cache coherent tiled Chip Multiprocessors (CMPs) ranging from 4 to 64 CPUs were used. The tiles are connected using a mesh configuration. Each tile comprises one core, Level 1 data and instruction caches, Level 2 unified cache, directory and router. A tile with SLAM hardware support comprises the same elements, with the addition of the corresponding SLAM hardware.

Two CMP configurations have been evaluated:

1. A CMP with private Level 1 and Level 2 caches. For this configuration, cache coherence is maintained using a directory-based SMP protocol.

2. A CMP with private Level 1 caches and shared Level 2 caches among all the CPUs in the CMP. For this configuration, cache coherence is maintained using a two-level directory-based protocol. This configuration constitutes a Non Uniform Cache Access (NUCA) architecture [43].

This allowed to evaluate SLAM’s performance using a configuration with lower cache hit latencies but smaller on-chip cache capacity (private L2 caches), as well as a configuration with higher cache hit latencies but larger on-chip cache capacity (shared L2 caches).
The simulation of the CPUs is performed using Simics [54]. This is done at a functional level in a full system simulation environment where the applications are run on top of an operating system. The memory hierarchy as well as the processor interconnect are simulated using cycle-level models using GEMS [55] and GARNET [64] respectively. SLAM hardware is modeled using a cycle-level model implemented using a Simics module. All the models are extensively instrumented in order to collect statistical data, as well as to evaluate performance.

The parameters of the baseline CMP (without SLAM hardware support) are listed in Table 5.3. The parameters of the SLAM hardware simulated are listed in Table 5.4. Block Size refers to the number of TS entries that are copied to or brought from the next level in the case of a 1st level TS cache miss. The proposed SLAM implementation comprises the baseline CMP with SLAM hardware support incorporated.

The sizes of the Task Stack and the Buffer of Stealable Tasks were defined after the evaluation of different size combinations. The chosen ones provided a good performance when working with fine grain applications. This with a relatively low cost in terms of hardware resources.

Additionally, in order to analyze Cilk’s performance, Cilk’s runtime system was instrumented. This instrumentation allows the measurement of values related to steal granularity when running Cilk applications. This functionality was added as a separate Simics module.

5.6 Results of CMPs with Private Level 2 Caches

5.6.1 Serial Evaluation

In order to get a measurement of the efficiency of SLAM related to a serial C version of the benchmarks tested, a performance evaluation was done using a single core machine executing the set of divide-and-conquer applications evaluated. The simulation is performed using a coherence protocol just as if the CPU was a CMP. This was with the objective of getting the corresponding overhead reflected in the results. The parameters of this baseline single core machine (without SLAM hardware support) are shown in Table 5.5. The parameters of the SLAM hardware simulated are the same as those shown in Table 5.4.

Figure 5.1 shows the execution time of single thread executions of SLAM and
### CHAPTER 5. SLAM EVALUATION

<table>
<thead>
<tr>
<th>OS</th>
<th>Linux kernel 2.6.15 (4 to 64 CPUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>x86</td>
</tr>
<tr>
<td>Coherence</td>
<td>Directory-based, MOESI, sequential consistency</td>
</tr>
<tr>
<td>Directory</td>
<td>1 bank per core, 6-cycle latency</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32 KB 4-way set associative split Data/Instruction 3-cycle latency private to each core</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512 KB 16-way set associative 6-cycle tag/6-cycle data latencies per core</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB 160-cycle latency shared among all the cores</td>
</tr>
<tr>
<td>Interconnect</td>
<td>2D Mesh with 1-cycle link latency between tiles, 16 flits per link</td>
</tr>
<tr>
<td>Links</td>
<td>16B flits, 1-cycle link latency</td>
</tr>
<tr>
<td>Router</td>
<td>4-stages pipeline, 4 VCs per Virtual Network, 4 flits buffer per VC, 4 Virtual Networks for private L2 caches, 3 Virtual Networks for shared L2 caches</td>
</tr>
</tbody>
</table>

| Task Stack | 64 entries 1-cycle latency |
| Buffer of Stealable Tasks | 32 entries 1-cycle latency |
| Replacement Block Size | 4 entries |

Table 5.3: Configuration of the baseline simulated system.

Table 5.4: Configuration of SLAM simulated systems.
Table 5.5: Configuration of the single core baseline system.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Linux kernel 2.6.15</td>
</tr>
<tr>
<td>CPU</td>
<td>x86</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32 KB 4-way set associative split Data/Instruction 3−cycle latency private to each core</td>
</tr>
<tr>
<td>L2 cache</td>
<td>16 MB 16-way set associative 6−cycle tag/ 6−cycle data latencies</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB 160−cycle latency shared among all the cores</td>
</tr>
</tbody>
</table>

Cilk versus the corresponding C version of the benchmarks evaluated. The data is normalized to the execution time of the C version for each case. The algorithms implemented are the same for both systems. *Nfib* was the benchmark with the worst performance, with SLAM 81% and Cilk 376% slower than C respectively. The very fine granularity of the tasks computed in this benchmark makes the creation of potentially parallel tasks a very expensive process for both, SLAM and Cilk. However, the fact that SLAM was less than two times slower than C, shows that it was capable of exploiting very fine grain parallelism with a reasonably low cost (average task granularity for *SLAM Nfib* is 17 to 21 simulation cycles, see Tables 5.14 and 5.15).

For the other benchmarks SLAM and Cilk results were very close to each other, with SLAM slightly faster in all except *Cholesky, LU Low* and *FFT High*. In the cases of *Cholesky* and *FFT High*, Cilk was even faster than C taking only 96% and 99% of the normalized time respectively. In *LU Low* both systems delivered the same performance being 29% slower than C. For all the other applications, SLAM was faster than Cilk with percentage differences ranging from 2 to 11% related to the normalized execution time. The reason for these results is that the task granularity is coarser in these applications. This fact tends to hide the cost of task creation in the overall execution time.

For the cases in which SLAM and Cilk were faster than C, the compiler (using -O2 as optimization level) generated slightly less efficient code for the C versions of the mentioned applications.
5.6.2 Parallel Evaluation

For the parallel evaluation of the SLAM implementation proposed in this work, the benchmarks were executed on CMPs using multiple threads. The number of SLAM/Cilk worker threads created by each application was equal to the number of cores available in the corresponding CMP. Each of these worker threads was statically pinned to one core in the system. Figure 5.2 shows the speedup\(^1\) of SLAM and Cilk versus C for the divide-and-conquer applications tested. Figure 5.3 shows the speedup of the loop-level parallel applications evaluated versus a single-thread C version for each case. All the experiments were run using the Hoard memory allocator v3.8 [11] in order to speed up concurrent dynamic memory allocations.

Some of the divide-and-conquer applications were tested using two different inputs, Low and High; Low being a relatively small input intended to generate a smaller amount of parallelism, leading to finer grain tasks. Conversely, High corresponds to an input which generates a bigger amount of parallelism, leading in turn to coarser grained tasks. These experiments allowed the evaluation of SLAM in situations in which parallelism is abundant, but also in situations in

\(^1\)This is absolute speedup since it is measured related to the execution time of the serial C version for each benchmark.
which it is limited.

The speedup was calculated using Equation 5.1, where $T_{\text{C version}}$ is the execution time of the corresponding C application using a single core, and $T_{\text{N CPUs}}$ is the execution time of that same SLAM or Cilk application using $N$ cores.

$$Absolute Speedup = \frac{T_{\text{C version}}}{T_{\text{N CPUs}}} \quad (5.1)$$

$Nfib$ was the benchmark with the biggest performance difference between SLAM and Cilk. For 32 cores, SLAM delivered a speedup of 14.84 versus 5.56 obtained with Cilk. In this case, since $Nfib$ tasks are very small, SLAM hardware support showed the capability of exploiting fine grain parallelism.

For $N Queens$ the results show that SLAM and Cilk had a very similar performance up to 16 cores. However, for the case of 32 cores, SLAM had a noticeable performance reduction in terms of scalability whilst Cilk keep scaling consistently\(^2\). When using 32 cores, SLAM was 37.02% slower than Cilk for the case of $N Queens Low$. A similar result was obtained for $N Queens High$, in which SLAM was 30.60% slower than Cilk. In these cases the Cilk load balancing system was able to make a better workload distribution than SLAM’s one.

For the case of $Merge Sort$, both systems showed poor results. Still, SLAM performed slightly better than Cilk. For the case of 16 cores, for which the results showed a positive speedup, SLAM was 6.42% faster than Cilk.

Executing $Quick Sort$, SLAM showed a consistently better performance than Cilk. Using 4 cores, SLAM was 7.83% faster than Cilk, whilst using 32 cores the difference was 38.80%.

Looking at the case of $LU Low$, SLAM and Cilk delivered a very similar performance up to 8 cores. However, when using 16 and 32 cores, SLAM clearly had a better performance, being 38.57% and 278.73% faster than Cilk respectively. It can be seen in the graph that SLAM performance scaled from 4 to 32 cores, whilst Cilk did not scale beyond 16 cores.

Evaluating $FFT Low$, the performance delivered by SLAM and Cilk was practically identical for the cases of 4 and 8 cores. When using 16 cores SLAM was

\(^2\)For this particular case the workload distribution performed by SLAM’s runtime system produced a lower data locality than its Cilk counterpart.
Figure 5.2: Speedup of SLAM and Cilk versus C for the divide-and-conquer parallel benchmarks tested using private Level 2 caches.
faster than Cilk by 20.35%. For the 32-core configuration, SLAM was still able to get speedup, whilst Cilk got a negative speedup compared with its 16-core case. When using 32 cores SLAM was 74.54% faster than Cilk. In the case of FFT High, again both systems delivered very similar performances with 4 and 8 cores. However, again SLAM started to show better scalability when using 16 and 32 cores, being faster than Cilk by 6.74% and 14.11% respectively.

The experiments done for Cholesky showed a limited scalability in both systems. However, SLAM got speedup in all the configurations, even when it was limited. On the other hand, Cilk was not able to scale beyond 8 cores. The results showed SLAM being faster than Cilk by 7.87% when using 4 cores. Using the 32-core configuration, SLAM was 379.33% faster than Cilk.

Heat delivered speedup in both systems for all the configurations evaluated. Again the results show that SLAM and Cilk had very similar performances with 4 and 8 cores. From 16 to 32 cores, SLAM showed a good scalability, being faster than Cilk by 10.47% when using 16 cores, and by 61.32% when using 32.

Matmult showed very similar speedups for SLAM and Pthreads, with a more noticeable difference when using 32 cores. In this case SLAM was 11.74% faster than the Pthreads implementation.

Integration showed practically the same speedups for both, SLAM and the Pthreads implementation.

For the cases of LU Low, FFT Low and Cholesky, from 16 cores onwards Cilk showed a lower performance as more cores were added to the system. On the other hand, SLAM kept showing speedups with a higher number of cores. A similar effect was observed also when executing Heat, for which Cilk practically did not show speedup when using 32 cores relative to the 16-core case. The
fact that the input used for these benchmarks is small (which means that the available parallelism is limited) makes the overhead associated with scheduling operations more visible for both systems. The results corresponding to the above benchmarks show that SLAM’s scheduling overhead is much lower than Cilk’s. This is because unlike Cilk, SLAM does not make use of the shared memory when distributing the workload among the cores performing the computation. This produces a higher data locality, which in turn results in better performance.

5.6.3 Stealing and Running Time Distribution

In order to get an estimation of SLAM’s work stealing mechanism efficiency, measures of the stealing time spent by each CPU were taken while executing each application using multiple threads. These measurements were also taken for Cilk applications. In this way it was possible to have a view of the work stealing mechanism efficiency of each system.

The stealing time in this case is the time a thief CPU spends in stealing a task from a victim. It is counted from the beginning of a steal process until the end, that is to say, when the thief gets a task from a victim. It can be the case that a thief has to try several times before being able to steal some work. This is due to the situation that some victims might not have exportable tasks at the moment of the steal.

There also might be a case in which a CPU tries to steal some work, but there are no more potentially parallel tasks available in any other CPU. This situation arises when there is one CPU, or a group of them, finishing the execution of tasks that do not generate further parallelism, in consequence, all the other CPUs in the system are trying to steal. In this case some CPU must signal the end of the corresponding parallel region so that the other CPUs stop their stealing processes. When a situation like this happens, the stealing time is counted from the beginning of the stealing process, until the arrival of the signal indicating the end of the parallel region.

In this way, the total stealing time a CPU spends when executing an application, is the sum of the times of all the stealing processes it performed during the execution. Figure 5.4 shows a breakdown of the execution time of the benchmarks tested using SLAM and Cilk\(^3\).

\(^3\)The times shown in the graphs are average values of all the times spent by each CPU for each corresponding experiment.
The times shown are normalized to the execution time of the appropriate Cilk application for each case. The elements illustrated in the graphs are:

1. The running time, which is the time both systems spent computing tasks from the application.

2. The stealing time, which is the time a CPU spent stealing and/or trying to steal work from remote CPUs.

The graphs for $Nfib$ show that this application keeps the CPUs busy almost all of the time. Only for the cases of 16 and 32 CPUs executing Cilk, the stealing time started to grow. When using 16 cores, Cilk’s average stealing time per core was 0.36%; for 32 cores, this time was 0.78%.

$NQueens\text{ Low}$ showed a situation similar to the previous case. For this application the stealing times were almost zero, except for the cases of 16 and 32 cores executing Cilk applications. In these, the average stealing times per CPU were 0.33% and 1.32% respectively. $NQueens\text{ High}$ produced much more parallelism so that, in this case the only noticeable stealing time appeared when 32 cores where executing Cilk, being 0.24%.

In $Merge\ Sort$, the stealing times were all very significant when executing SLAM applications. The percentages were 16.63%, 27.85%, 29.94% and 21.67% using 4, 8, 16 and 32 cores respectively. On the other hand, Cilk only had a visible stealing time when using 32 cores with 0.37% of the total execution time.

Again, $Quick\ Sort$ showed much longer stealing times when evaluating SLAM applications. The percentages were 12.38%, 24.42, 32.62% and 29.69% when using 4, 8, 16 and 32 cores respectively. On the other hand, Cilk stealing times were visible when using 8, 16 and 32 cores with percentages of 0.12%, 0.64% and 1.35% respectively.

For the case of $LU\text{ Low}$, the stealing times of SLAM were in the range from 0.56% and 3.57%, showing the maximum percentage when using 16 cores. Cilk stealing times percentages were in the range of 0.34% and 1.58% reaching the maximum value when using 32 cores.

In the case of $FFT$, the stealing times of both systems were closer to each other from 8 cores onwards. SLAM showed a range from 0.95% to 2.74%, whilst Cilk’s range was from 0.66% to 2.03%. Both systems reached their maximum values when using 32 cores.
Figure 5.4: Stealing and running time distribution for the configurations with private Level 2 caches. $C$ and $S$ below the bars correspond to Cilk and SLAM respectively.
When evaluating Cholesky, the stealing times were visible in all the configurations for both systems. SLAM cores had a stealing time ranging from 4.44% to 9.21% reaching its peak when using 16 cores. Cilk, on the other hand, showed stealing times from 0.90% to 1.89%, showing the maximum value when using 16 cores.

Finally in Heat, the stealing times were again visible in all the configurations for SLAM. The values were in the range of 0.36% and 3.45%, with the maximum value reached using 32 cores. Cilk stealing times were visible from 8 cores onwards. The range of values was from 0.12% to 3.33% reaching its peak at 32 cores.

An interesting situation is shown by these results. SLAM stealing times are consistently longer than Cilk ones, a situation that is emphasized when executing the sorting applications. This situation is explained by the way each system handles synchronization points during program execution.

Cilk uses the \texttt{sync} statement [28], which acts as a local barrier where a thread waits until the result of a previously spawned Cilk procedure has been computed. The corresponding thread is able to proceed only when this result is ready, even if the CPU is doing nothing while waiting for it.

In SLAM, as explained in Section 4.5 of Chapter 4, the way in which synchronization points are handled is setting \textit{Steal} tasks in the Task Stack entries whose tasks have been stolen by a remote CPU. When a SLAM thread reaches a synchronization point, it executes the \textit{Steal} task planted previously leading it to try to get some work from a victim.

In this way, the fact that SLAM threads do not remain \textit{idle} when they reach a synchronization point in the execution of a program, leads the involved CPUs to spend longer time stealing, trying in this way to keep the CPUs busy as much time as possible. This situation is clearly reflected in the graphs shown in Figures 5.4 and 5.9. These graphs show that even though the stealing times of SLAM and Cilk workers are comparable, and sometimes higher for SLAM workers, the total execution time favours in general the SLAM system.

In the presence of enough parallelism, the mechanism that makes a SLAM worker thread to try to get some work from another processing node when it reaches a synchronization point, can result in a faster distribution of the workload of a given computation, which in turn can potentially lead to a higher overall performance.
5.6.4 Steal Granularity Information

In the evaluation of the system proposed in this work, measurements of the *steal granularity* were taken. These measurements were useful to get an idea of SLAM’s dynamic load balancing efficiency as a whole. Cilk’s runtime system was also evaluated under this criterion, allowing in this way to make the comparison between both systems.

*Steal Granularity* in this context is defined as the number of cycles a CPU executes after it has stolen a task from a victim and before it has to steal again. In SLAM’s case, a CPU begins a new stealing process when either it runs out of work, or it reaches a synchronization point at which the completion of a stolen task is expected. On the other hand, a CPU executing a Cilk application begins a new stealing process when it runs out of work in its local pool of tasks.

In general, a steal can be considered more or less profitable depending on the generated *steal granularity*; the bigger this is, the more profitable the steal. Profitable steals lead to less stealing in the system which, in turn leads to shorter time performing load balancing activities. Tables 5.6 and 5.7 show the average number of steal attempts and successful steals made per core during the execution of each application. The average values of the *Average, Maximum* and *Minimum steal granularities* generated per core, are shown in the Tables 5.8 and 5.9.

Figure 5.5 shows the average steal granularity distribution per core measured for the benchmarks evaluated. Each graph indicates the percentages of the total number of steals that lead to the different steal granularities shown.

Although the graphs show three different granularity values, the description of the results make reference only to two of them:

1. below 10K cycles.
2. equal or above 10K cycles.

The reason for this is to emphasize that steals leading to the execution of less than 10K cycles are more expensive, and in consequence, the less of this kind of steals in the execution of an application, the better. This applies for both, SLAM and Cilk.

In the results corresponding to *Nfib*, 75% of SLAM steals produced granularities equal or above 10K cycles when using 4 cores, going up to 85% when using 32. Cilk percentages on the other hand, went from 50% with 4 cores, to 58% for the 32 cores configuration.
### Table 5.6: Steals attempted (A) and made (M) running SLAM applications using CMPs with private Level 2 caches.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfib</td>
<td>1,688</td>
<td>26</td>
<td>2,096</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>2,097</td>
<td>39</td>
<td>2,697</td>
<td>45</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>557</td>
<td>22</td>
<td>1,267</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3,448</td>
<td>38</td>
<td>3,629</td>
<td>40</td>
</tr>
<tr>
<td>NQueens High</td>
<td>558</td>
<td>26</td>
<td>913</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>4,801</td>
<td>37</td>
<td>4,926</td>
<td>44</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>5.5x10^5</td>
<td>1</td>
<td>9.2x10^5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>9.9x10^5</td>
<td>47</td>
<td>10.0x10^6</td>
<td>166</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>4.4x10^6</td>
<td>39</td>
<td>8.0x10^6</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>9.2x10^6</td>
<td>5,062</td>
<td>8.5x10^6</td>
<td>9,926</td>
</tr>
<tr>
<td>LU Low</td>
<td>133,047</td>
<td>334</td>
<td>363,696</td>
<td>1,322</td>
</tr>
<tr>
<td></td>
<td>484,076</td>
<td>1,639</td>
<td>663,867</td>
<td>1,833</td>
</tr>
<tr>
<td>FFT Low</td>
<td>34,999</td>
<td>35</td>
<td>70,213</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>57,543</td>
<td>144</td>
<td>146,816</td>
<td>293</td>
</tr>
<tr>
<td>FFT High</td>
<td>97,542</td>
<td>127</td>
<td>145,004</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>134,704</td>
<td>144</td>
<td>170,016</td>
<td>345</td>
</tr>
<tr>
<td>Cholesky</td>
<td>135,051</td>
<td>6,676</td>
<td>1.2x10^6</td>
<td>19,944</td>
</tr>
<tr>
<td></td>
<td>1.6x10^6</td>
<td>35,639</td>
<td>2.1x10^6</td>
<td>40,492</td>
</tr>
<tr>
<td>Heat</td>
<td>494,366</td>
<td>376</td>
<td>102,903</td>
<td>492</td>
</tr>
<tr>
<td></td>
<td>106,328</td>
<td>503</td>
<td>130,753</td>
<td>430</td>
</tr>
</tbody>
</table>

### Table 5.7: Steals attempted (A) and made (M) running Cilk applications using CMPs with private Level 2 caches.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfib</td>
<td>646</td>
<td>25</td>
<td>405</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>1,033</td>
<td>35</td>
<td>4,829</td>
<td>35</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>194</td>
<td>26</td>
<td>797</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>1,587</td>
<td>51</td>
<td>2,603</td>
<td>34</td>
</tr>
<tr>
<td>NQueens High</td>
<td>216</td>
<td>35</td>
<td>523</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>1,826</td>
<td>45</td>
<td>5,046</td>
<td>55</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>4.2x10^5</td>
<td>2</td>
<td>4.3x10^5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3.5x10^5</td>
<td>16</td>
<td>3.1x10^6</td>
<td>15</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>7.3x10^5</td>
<td>102</td>
<td>7.8x10^5</td>
<td>295</td>
</tr>
<tr>
<td></td>
<td>6.2x10^6</td>
<td>1,127</td>
<td>5.7x10^6</td>
<td>1,673</td>
</tr>
<tr>
<td>LU Low</td>
<td>1.3x10^6</td>
<td>526</td>
<td>1.6x10^6</td>
<td>758</td>
</tr>
<tr>
<td></td>
<td>1.5x10^6</td>
<td>837</td>
<td>1.8x10^6</td>
<td>817</td>
</tr>
<tr>
<td>FFT Low</td>
<td>16,620</td>
<td>65</td>
<td>15,636</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>14,969</td>
<td>90</td>
<td>21,480</td>
<td>87</td>
</tr>
<tr>
<td>FFT High</td>
<td>34,290</td>
<td>69</td>
<td>32,014</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>26,455</td>
<td>116</td>
<td>30,188</td>
<td>111</td>
</tr>
<tr>
<td>Cholesky</td>
<td>60,684</td>
<td>1,656</td>
<td>80,928</td>
<td>1,825</td>
</tr>
<tr>
<td></td>
<td>142,397</td>
<td>2,428</td>
<td>295,142</td>
<td>1,901</td>
</tr>
<tr>
<td>Heat</td>
<td>44,295</td>
<td>1,012</td>
<td>29,852</td>
<td>1,236</td>
</tr>
<tr>
<td></td>
<td>33,035</td>
<td>1,193</td>
<td>53,216</td>
<td>1,016</td>
</tr>
</tbody>
</table>
### SLAM Steal Granularity

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfib</td>
<td>73.137</td>
<td>25.665</td>
<td>14.497</td>
<td>7.023</td>
<td>3,374.844</td>
<td>1,924.150</td>
<td>744.691</td>
<td>293.551</td>
<td>434.884</td>
<td>257.521</td>
<td>140.094</td>
<td>94.323</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>85.839</td>
<td>35.391</td>
<td>11.862</td>
<td>9.646</td>
<td>395.391</td>
<td>297.981</td>
<td>239.347</td>
<td>145.464</td>
<td>4,716</td>
<td>5,367</td>
<td>5,238</td>
<td>6,724</td>
</tr>
<tr>
<td>NQueens High</td>
<td>463.040</td>
<td>155.974</td>
<td>80.857</td>
<td>48.310</td>
<td>2,333.048</td>
<td>1,901.085</td>
<td>1,409.730</td>
<td>902.372</td>
<td>7,326</td>
<td>5,206</td>
<td>5,230</td>
<td>5,769</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>867.752</td>
<td>114.002</td>
<td>19.178</td>
<td>14.309</td>
<td>956.375</td>
<td>594.201</td>
<td>555.536</td>
<td>448.498</td>
<td>1,275</td>
<td>1,672</td>
<td>2,047</td>
<td>2,320</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>53.096</td>
<td>2.112</td>
<td>0.167</td>
<td>0.071</td>
<td>598.362</td>
<td>154.245</td>
<td>96.285</td>
<td>58.074</td>
<td>1,496</td>
<td>387</td>
<td>173</td>
<td>185</td>
</tr>
<tr>
<td>LU Low</td>
<td>3.613</td>
<td>0.561</td>
<td>0.284</td>
<td>0.239</td>
<td>334.667</td>
<td>50.007</td>
<td>22.679</td>
<td>6.019</td>
<td>434.038</td>
<td>10.165</td>
<td>12.211</td>
<td>10.605</td>
</tr>
<tr>
<td>FFT Low</td>
<td>13.863</td>
<td>5.991</td>
<td>2.724</td>
<td>0.460</td>
<td>372.800</td>
<td>261.556</td>
<td>157.725</td>
<td>337.250</td>
<td>7,220</td>
<td>9,270</td>
<td>10,304</td>
<td>10,150</td>
</tr>
<tr>
<td>FFT High</td>
<td>21.060</td>
<td>2.938</td>
<td>2.091</td>
<td>0.460</td>
<td>344.667</td>
<td>40.007</td>
<td>15.226</td>
<td>6.019</td>
<td>34,048</td>
<td>10,165</td>
<td>12,211</td>
<td>10,605</td>
</tr>
<tr>
<td>Cholesky</td>
<td>0.228</td>
<td>0.051</td>
<td>0.024</td>
<td>0.020</td>
<td>286.467</td>
<td>65.567</td>
<td>22.679</td>
<td>31.522</td>
<td>478,037</td>
<td>159,557</td>
<td>180,238</td>
<td>291,370</td>
</tr>
</tbody>
</table>

Table 5.8: Average, Maximum and Minimum steal granularities running SLAM applications using CMPs with private Level 2 caches. The values are given in millions of simulation cycles for the cases of Average and Maximum and in simulation cycles in the case of Minimum.

### Cilk Steal Granularity

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfib</td>
<td>200.823</td>
<td>107.758</td>
<td>30.084</td>
<td>20.867</td>
<td>2,894.536</td>
<td>1,201.577</td>
<td>650.471</td>
<td>328.561</td>
<td>2,440</td>
<td>2,923</td>
<td>3,992</td>
<td>4,943</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>62.474</td>
<td>27.464</td>
<td>15.038</td>
<td>7.625</td>
<td>471.480</td>
<td>309.510</td>
<td>141.394</td>
<td>61.246</td>
<td>4,679</td>
<td>6,465</td>
<td>13,460</td>
<td>7,551</td>
</tr>
<tr>
<td>NQueens High</td>
<td>300.343</td>
<td>188.670</td>
<td>64.490</td>
<td>27.570</td>
<td>2,923.078</td>
<td>2,032.738</td>
<td>517.172</td>
<td>269.916</td>
<td>4,814</td>
<td>5,763</td>
<td>6,257</td>
<td>6,614</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>21.740</td>
<td>6.026</td>
<td>1.270</td>
<td>0.375</td>
<td>661.473</td>
<td>153.819</td>
<td>222.228</td>
<td>167.225</td>
<td>2,305</td>
<td>2,215</td>
<td>2,515</td>
<td>3,125</td>
</tr>
<tr>
<td>LU Low</td>
<td>2.261</td>
<td>1.035</td>
<td>0.798</td>
<td>1.181</td>
<td>385.441</td>
<td>66.153</td>
<td>27.528</td>
<td>45.839</td>
<td>2,108</td>
<td>3,143</td>
<td>2,378</td>
<td>2,918</td>
</tr>
<tr>
<td>FFT Low</td>
<td>12.128</td>
<td>5.153</td>
<td>2.292</td>
<td>3.115</td>
<td>124.626</td>
<td>86.615</td>
<td>49.602</td>
<td>46.133</td>
<td>2,152</td>
<td>2,488</td>
<td>2,470</td>
<td>3,119</td>
</tr>
<tr>
<td>Cholesky</td>
<td>0.874</td>
<td>0.372</td>
<td>0.182</td>
<td>2.181</td>
<td>1,317</td>
<td>2,458</td>
<td>2,758</td>
<td>3,255</td>
<td>1,874</td>
<td>2,458</td>
<td>2,758</td>
<td>3,255</td>
</tr>
</tbody>
</table>

Table 5.9: Average, Maximum and Minimum steal granularities running Cilk applications using CMPs with private Level 2 caches. The values are given in millions of simulation cycles for the cases of Average and Maximum and in simulation cycles in the case of Minimum.
Figure 5.5: Average steal granularity distribution with private Level 2 caches.
For *NQueens* it can be observed that relatively low percentages of steals lead to granularities below 10K in both systems. In SLAM’s case, the values fell between 13% and 6% for the case of the *low* input. With the *high* input, the values were between 9% and 4%. In Cilk’s case, the steal percentages that produced granularities smaller than 10K cycles were between 15% and 12%, for the case of the *low* input. When using the *high* input, the values fell within 16% and 12%.

For *Merge Sort*, the graphs show that only Cilk steals led to granularities below 10K cycles; this when using from 4 cores onwards. SLAM steals on the other hand only produced granularities equal to or above 10K cycles in all the cases.

The *Quick Sort* results show a different tendency from the previous cases when using from 8 cores onwards. For this case, higher percentages of Cilk steals led to granularities equal or above 10K cycles compared with the corresponding SLAM values. For this benchmark, Cilk steal percentages that led to granularities below 10K cycles went from 41% to 13% when using 4 and 32 cores respectively, decreasing as more cores were used. Conversely, SLAM values for the same granularities went from 26% to 58% when using 4 and 32 cores respectively, increasing as more cores were used.

The results of *LU Low* show that all the SLAM steals led to granularities equal to or above 10K cycles. On the other hand, some Cilk steals produced granularities below 10K cycles. The percentages of the above steals were in the range from 25% and 13% with 4 and 32 cores respectively. Again, the more cores that were used, the lower the value of the mentioned percentages.

*FFT* produced similar results for both, the *low* and *high* inputs. For the *low* input SLAM steal percentages producing granularities below 10K cycles were nearly zero. Cilk steals on the other hand, produced granularities below 10K cycles in all the configurations. The values fell between 53% and 38%, decreasing as more cores were used in the system. The case of the *high* input was similar, but in this case, 4% and 1% of SLAM steals produced below 10K cycles granularities when using 4 and 8 cores respectively, again the other cases were almost zero. On the other hand, Cilk steals produced less than 10K cycles granularities with values that fell between 53% to 40%, decreasing as more cores were added to the system.

*Cholesky* results showed that a higher percentage of SLAM steals lead to below 10K cycles granularities related to the corresponding Cilk ones. In SLAM’s case
the values fell between 34% and 29%. Cilk values on the other hand, went from 20% to 9% decreasing as more cores were used in the system.

Finally Heat delivered results showing that SLAM steals only produced granularities equal to or above 10K cycles for all the configurations. On the other hand, Cilk steals led to less than 10K cycles granularities with values that fell between 54% to 34%, decreasing as more cores were used in the system.

It can be observed that in general, SLAM successful steals produced larger steal granularities than those corresponding to Cilk. This larger steal granularity contributes to have a smaller overhead associated with load balancing operations, which contributes to get a better overall performance.

5.7 Results of CMPs with Shared Level 2 Caches

5.7.1 Serial Evaluation

As with the CMPs with private L2 caches, this performance evaluation was done using a single core machine executing the set of divide-and-conquer applications used. The coherence protocol used in this case is the NUMA [43] protocol used for simulating the CMPs with shared L2 caches. The parameters of this baseline single core machine (without SLAM hardware support) are shown in Table 5.5. The parameters of the SLAM hardware simulated are the same as those shown in Table 5.4.

Figure 5.6 shows the execution time of single-thread executions of SLAM and Cilk versus the corresponding C version of the benchmarks evaluated. The values shown are normalized to the execution time of the C version for each case. Again Nfib was the benchmark with the worst performance, with SLAM 76% and Cilk 360% slower than C respectively.

For the case of the other benchmarks the performance of both system was very similar. SLAM was slower than Cilk for NQueens by a 1% difference for the case of the low input (the percentage differences are related to the normalized execution time). However, for the case of the high input, Cilk was slower than SLAM with a difference of 3%. Cholesky again showed SLAM being slower than Cilk by a 6% difference.

For the case of LU Low SLAM and Cilk were slower than C by 25%. However LU High showed SLAM and Cilk being faster than C by a 1% difference. In the
remaining applications SLAM was faster than Cilk with a difference of 9% when executing *Merge Sort*, 2% in *Quick Sort* and *FFT Low*, 1% in *FFT High* and by 3% in *Heat’s* case.

**Figure 5.6**: Execution time of single thread execution of SLAM and Cilk versus C.

### 5.7.2 Parallel Evaluation

In this evaluation, CMPs ranging from 4 to 64 cores were used. The benchmarks used for this evaluation were the same as the ones used when evaluating the CMPs with private Level 2 caches, plus the inclusion of *LU High*. For these experiments *NQueens Low, Merge Sort, Quick Sort, LU Low, FFT Low* and *Cholesky* were run using from 4 to 32 threads. Similarly, *Nfib, NQueens High, LU High, FFT High, Heat, Matmult* and *Integration* were run using from 4 to 64 threads. For all the cases, the number of SLAM/Cilk worker threads created by the application was equal to the number of cores available in the corresponding configuration. Each of these worker threads was statically pinned to one core in the system.

Figure 5.7 shows the speedup of SLAM and Cilk versus C for the set of divide-and-conquer applications evaluated. Figure 5.8 shows the speedup of the loop-level parallel applications evaluated. The speedup was calculated using
Equation 5.1, where $Time_{C\text{-version}}$ is the execution time of the corresponding C application using a single core, and $Time_{N\text{-CPUs}}$ is the execution time of the same SLAM or Cilk application using $N$ cores.

$Nfib$ was again the benchmark that showed the biggest difference in performance between SLAM and Cilk. This, as explained before, was due to the fine granularity of the tasks executed. The results showed SLAM being faster than Cilk within a range from 166.43% using 4 cores, to 217.19% for the 64-core case.

When running $NQueens$ with the $Low$ and $High$ inputs, the performance delivered by SLAM and Cilk was nearly equal from 4 to 16 cores. With 32 cores, SLAM was slower than Cilk by 26.14% and 18.47% for the $Low$ and $High$ inputs respectively. However when executing $NQueens High$ using 64 cores, SLAM was faster than Cilk by 3.17%.

For the case of $Merge\ Sort$, the performance delivered by both systems was very irregular in terms of scalability. The results showed that SLAM was slightly faster than Cilk in all the cases, however the speedup was very poor. The maximum speedup obtained was 1.96 using 16 cores, SLAM being faster than Cilk by 3.3%.

The performance delivered by SLAM and Cilk when executing $Quick\ Sort$ was again poor in terms of scalability. The maximum speedup obtained was 2.50 for the case of SLAM and 2.49 for the case of Cilk, showing practically identical performances when using 32 cores.

$LU$ showed favourable results for SLAM for both, the $Low$ and $High$ inputs. When running $LU Low$, Cilk was competitive delivering almost the same performance as SLAM when using 4 and 8 cores. However, for the cases of 16 and 32 cores, SLAM was faster by 21.58% and 274% respectively, with Cilk not scaling beyond 16 cores. For the case of $LU High$, the situation was similar, with both systems delivering nearly identical performances from 4 to 16 cores. But again, when using 32 and 64 cores, SLAM was faster by 21.78% and 265.25% respectively, with Cilk not scaling beyond 32 cores.

The experiments for $FFT\ Low$ showed that SLAM was slightly slower than Cilk for the 4 and 8 core cases, which can be seen in the corresponding speedup values showed in the graph. However, with the addition of more cores, SLAM delivered better performance. When using 16 cores, SLAM was faster by 5.40% whilst for the 32 cores case, it was faster by 28.25%. For $FFT\ High$ the results show SLAM being slower than Cilk when using 16 and 32 cores by 12.47% and 10.16% respectively. However when using 64 cores, Cilk’s performance went
Figure 5.7: Speedup of SLAM and Cilk versus C for the divide-and-conquer parallel benchmarks tested using shared Level 2 caches.
down drastically whilst SLAM kept scaling consistently, being faster than Cilk by 454.37%.

The experiments for Cholesky showed SLAM being faster than Cilk in all the cases. However, the speedup was not very large and neither SLAM nor Cilk scaled beyond 16 cores. The maximum speedup was obtained using 16 cores. For this case SLAM was faster by 29.68%.

Heat results showed SLAM delivering better performance than Cilk for all the cases evaluated. Similar to other benchmarks, there was a point beyond which Cilk was unable to keep scaling. This point was 32 cores for this benchmark. On the other hand, SLAM scaled consistently in all the core counts evaluated. When using 64 cores, SLAM was faster than Cilk by 496.75%. When using 16 cores (the last evaluation for which Cilk showed a positive speedup), SLAM was faster by 77.96%.

Matmult showed a very similar performance using both, SLAM and Pthreads. The biggest difference was when using 64 cores, in which SLAM was 6.24% faster than the Pthreads implementation.

Integration showed practically the same speedups for both, SLAM and the Pthreads implementation when using from 4 to 32 cores. However when using 64 cores SLAM turned out to be 14.52% faster. This could have been due to the small amount of computation performed, which in turn made the Pthreads scheduling overhead, as well as the one produced by the Operating System more visible.

As with the CMPs with private L2 caches, for LU Low, FFT Low and Cholesky Cilk showed low or even negative speedups from 16 cores onwards. The same effect was observed for LU High, FFT High and Heat, but from 32 cores onwards.
On the other hand, SLAM showed consistent speedups for the above applications, with the exception of Cholesky, which when using 32 cores showed a lower performance than when using 16. The reason for which Cilk applications with high inputs showed a performance degradation when using more than 32 cores is that, with a higher number of cores, the granularity of the executed tasks per core gets reduced. In consequence, the overhead associated with scheduling operations becomes more visible. The fact that Cilk uses the shared memory for performing communication operations related to scheduling, produces a lower data locality which, in turn results in lower performance.

5.7.3 Stealing and Running Time Distribution

The stealing times corresponding to the configurations with shared Level 2 caches are shown in Figure 5.9. The values are normalized to the execution times of the corresponding Cilk application for each case.

In Nfib, the stealing time was visible only when using Cilk from 16 cores onwards. The percentage values were 0.19%, 0.92% and 1.74% for 16, 32 and 64 cores respectively.

For the NQueens case, again the times were visible only when using Cilk. When running with the Low input, the values were 0.23% when using 16 cores and 0.95% for the 32 cores case. When running with the High input, the values were 0.18% and 1.60% when using 32 and 64 cores respectively.

When evaluating Merge Sort, only SLAM showed visible stealing times in the graphs. The values were in the range from 14.67% to 36.74%, reaching the maximum when using 16 cores.

The results for Quick Sort showed again relatively high values when using SLAM. The percentages fell between 12.04% and 44.08%, having its peak when using 32 cores. On the other hand, Cilk only showed visible values from 8 cores onwards. These were in the range from 0.11% to 0.95%, reaching the maximum value when using 32 cores.

The LU results showed SLAM having times with percentages from 0.43% to 3.50% when running the Low input, with the maximum value when using 32 cores. In the case of the High input, the values went from 0.13% to 3.29%, having its peak when using 32 cores. On the other hand, Cilk stealing times were visible from 8 cores onwards. The values were in the range from 0.16% to 0.91%, showing the highest one when using 16 and 32 cores, in the case of the Low input.
When running with the *High* input, the percentages fell between 0.13% to 0.63% showing the maximum when using 32 cores.

When evaluating *FFT*, SLAM values were in the range from 0.15% to 2.90% showing the peak when using 32 cores. Cilk on the other hand, showed values visible only from 8 cores onwards falling between 0.12% to 1.37% reaching the highest value also when using 32 cores.

*Cholesky* results showed visible stealing times in all the configurations for both systems. SLAM times were in the range from 2.70% to 6.09%, showing a peak when using 16 cores. Cilk values fell in the range from 0.54% to 1.49%, reaching a peak when using also 16 cores.

Finally *Heat* results showed SLAM visible stealing times for all the configurations. The values fell in the range from 0.40% to 11.70%, showing the maximum when using 32 cores. On the other hand, Cilk showed visible stealing times only from 8 cores onwards, with values from 0.05% to 2.68%, reaching a peak when using 32 cores.

It can be observed that SLAM and Cilk workers spend a very low proportion of their total execution time in stealing operations. In general SLAM workers spend a similar or larger amount of time stealing than their Cilk counterparts. However SLAM workers had a better performance in part due to the faster work load distribution achieved by SLAM’s load balancing system.

### 5.7.4 Steal Granularity Information

Tables 5.10 and 5.11 show the average number of steal attempts and successful steals made per core during the execution of each application. The average values of the *Average*, *Maximum* and *Minimum steal granularities* generated per core, are shown in the Tables 5.12 and 5.13.

Figure 5.10 shows the average steal granularity distribution per core for the benchmarks evaluated using CMPs with shared Level 2 caches. Each graph indicates the percentages of the total number of steals that led to the different steal granularities shown.

*Nfib* results show that just a few SLAM steals led to granularities below 10K cycles. The percentages fell in the range from 15% to 34%. Cilk steals percentages on the other hand, were considerably higher for the same granularity. The values were between 57% to 48% following a decreasing tendency when using from 4 to 32 cores, keeping a very similar value when jumping from 32 to 64.
Figure 5.9: Stealing and running time distribution for the configurations with shared Level 2 caches. C and S below the bars correspond to Cilk and SLAM respectively.
Figure 5.10: Average steal granularity distribution with shared Level 2 caches.
The results corresponding to *NQueens Low* show relatively small percentages of steals that led to less than 10K cycles granularities. In SLAM’s case, the values were between 13% and 8%. Cilk values were in the range from 19% to 15%. For the case of *NQueens High*, the situation was similar, showing again low percentages of steals leading to below 10K cycles granularities. In this case SLAM values were in the range from 19% and 8%, whilst Cilk values fell between 20% and 16%.

For the *Merge Sort* case, the results show that SLAM steals produced granularities equal to or above 10K cycles for all the configurations. Cilk steals on the other hand, produced less than 10K cycles granularities when using 16 and 32 cores, the percentages being 35% and 19% respectively.

*Quick Sort* showed results with high percentages of steals producing below 10K cycles granularities in the case of SLAM. The values were between 35% and 71%, following an increasing tendency as more cores were used. On the other end, Cilk values for the same granularity fell in the range from 47% to 25% following a decreasing tendency as more cores were used in the system.

*LU Low* results showed that SLAM steals only led to granularities equal or above 10K cycles in all the configurations. Cilk steals on the other hand, produced below 10K cycles granularities with percentages located between 34% and 19%, which followed a decreasing tendency as more cores were used. *LU High* delivered similar results in the sense that SLAM steals again only produced granularities equal to or above 10K cycles. Similarly, Cilk steals led to less than 10K cycles granularities with percentages located between 35% and 22%, values that were decreasing again as more cores were added to the system.

When evaluating *FFT Low*, SLAM steals produced less than 10K cycles granularities only with the 16 and 32 cores configurations; the percentages were 16% in both cases. On the other hand, Cilk steals produced granularities in the same order in all the configurations. The values were between 49% and 44%. The results corresponding to *FFT High* showed that when using 4 cores, SLAM steals only produced granularities equal or above 10K cycles. On the other cases evaluated, less than 10K cycles granularities were generated; the values were between 35% and 32%. Cilk on the other hand, had steals that led to granularities smaller than 10K cycles, whose values were located in the range from 52% to 44%.

Finally *Heat* results showed that SLAM steals only led to granularities equal or above 10K cycles in all the configurations. Cilk steals on the other hand,
produced less than 10K cycles granularities with percentages falling between 57% to 34%, following a decreasing tendency as more cores were added to the system.

As with the CMPs with shared Level 2 caches, in general SLAM successful steals produced larger granularities than those produced by Cilk successful steals. These larger granularities made SLAM successful steals to contribute to a lower load balancing overhead, which in turn led to a better overall performance.

5.8 A Deeper Look into SLAM

5.8.1 SLAM Relative Speedup

Figure 5.11 shows the relative speedup of SLAM when using CMPs with private Level 2 caches for the applications evaluated. Similarly, Figure 5.12 shows the relative speedup of SLAM using CMPs with shared Level 2 caches.

The speedup was calculated using Equation 5.2, where $Time_{1\text{-CPU}}$ is the SLAM execution time of the corresponding application using a single thread, and $Time_{N\text{-CPUs}}$ is the execution time of that same SLAM using $N$ threads.

$$Relative\text{Speedup} = \frac{Time_{1\text{-CPU}}}{Time_{N\text{-CPUs}}} \quad (5.2)$$

When using CMPs with private level 2 caches, the graph shows that only Cholesky, Merge Sort and Quick Sort delivered speedups below 5 when using 32 cores. On the other hand, with the exception of FFT Low, all the other applications delivered speedups above 10, scaling as more cores were added to the system.

In the case of the CMPs with shared level 2 caches, the results are similar in the sense that again Cholesky, Merge Sort and Quick Sort were the applications with low speedups when using 32 cores, in this case below 10. All the other applications showed speedups above 10 when using 32 cores, scaling consistently as more cores were added to the system. When using 64 core CMPs, all the applications evaluated showed a significant speedup related to their corresponding 32-core execution.

---

4The speedup is measured against the execution time of SLAM running with a single thread.
### Table 5.10: Steals attempted (A) and made (M) running SLAM applications using CMPs with shared Level 2 caches.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>M</td>
<td>A</td>
<td>M</td>
<td>A</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>670</td>
<td>20</td>
<td>1.081</td>
<td>27</td>
<td>3.036</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>1.3x10^5</td>
<td>1</td>
<td>2.95x10^6</td>
<td>4</td>
<td>27.4x10^9</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>5.1x10^8</td>
<td>33</td>
<td>1.25x10^10</td>
<td>880</td>
<td>1.48x10^10</td>
</tr>
<tr>
<td>LU Low</td>
<td>110,195</td>
<td>282</td>
<td>230,356</td>
<td>1,119</td>
<td>308,648</td>
</tr>
<tr>
<td>FFT Low</td>
<td>36,258</td>
<td>22</td>
<td>55,733</td>
<td>35</td>
<td>188,694</td>
</tr>
<tr>
<td>Cilk Steals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NQueens Low</td>
<td>278</td>
<td>36</td>
<td>495</td>
<td>42</td>
<td>1,249</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>2.0x10^6</td>
<td>2</td>
<td>3.0x10^7</td>
<td>9</td>
<td>1.0x10^11</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>829,149</td>
<td>33</td>
<td>1.0x10^8</td>
<td>680</td>
<td>805,306</td>
</tr>
<tr>
<td>LU Low</td>
<td>134,883</td>
<td>541</td>
<td>154,744</td>
<td>823</td>
<td>143,911</td>
</tr>
<tr>
<td>FFT Low</td>
<td>20,111</td>
<td>64</td>
<td>20,538</td>
<td>74</td>
<td>16,334</td>
</tr>
<tr>
<td>Cholesky</td>
<td>73,632</td>
<td>1,756</td>
<td>88,692</td>
<td>2,119</td>
<td>129,673</td>
</tr>
<tr>
<td>Nfib</td>
<td>632</td>
<td>36</td>
<td>414</td>
<td>33</td>
<td>1.265</td>
</tr>
<tr>
<td>NQueens High</td>
<td>176</td>
<td>34</td>
<td>678</td>
<td>44</td>
<td>1,174</td>
</tr>
<tr>
<td>FFT High</td>
<td>34,108</td>
<td>61</td>
<td>35,362</td>
<td>95</td>
<td>28,512</td>
</tr>
<tr>
<td>LU High</td>
<td>604,136</td>
<td>1,130</td>
<td>594,429</td>
<td>1,826</td>
<td>511,886</td>
</tr>
<tr>
<td>Heat</td>
<td>57,986</td>
<td>912</td>
<td>58,912</td>
<td>1,223</td>
<td>58,347</td>
</tr>
</tbody>
</table>

### Table 5.11: Steals attempted (A) and made (M) running Cilk applications using CMPs with shared Level 2 caches.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>M</td>
<td>A</td>
<td>M</td>
<td>A</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>218</td>
<td>36</td>
<td>495</td>
<td>42</td>
<td>1,249</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>1.2x10^6</td>
<td>2</td>
<td>1.2x10^7</td>
<td>9</td>
<td>1.0x10^11</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>829,149</td>
<td>33</td>
<td>1.0x10^8</td>
<td>680</td>
<td>805,306</td>
</tr>
<tr>
<td>LU Low</td>
<td>134,883</td>
<td>541</td>
<td>154,744</td>
<td>823</td>
<td>143,911</td>
</tr>
<tr>
<td>FFT Low</td>
<td>20,111</td>
<td>64</td>
<td>20,538</td>
<td>74</td>
<td>16,334</td>
</tr>
<tr>
<td>Cholesky</td>
<td>73,632</td>
<td>1,756</td>
<td>88,692</td>
<td>2,119</td>
<td>129,673</td>
</tr>
<tr>
<td>Nfib</td>
<td>632</td>
<td>36</td>
<td>414</td>
<td>33</td>
<td>1.265</td>
</tr>
<tr>
<td>NQueens High</td>
<td>176</td>
<td>34</td>
<td>678</td>
<td>44</td>
<td>1,174</td>
</tr>
<tr>
<td>FFT High</td>
<td>34,108</td>
<td>61</td>
<td>35,362</td>
<td>95</td>
<td>28,512</td>
</tr>
<tr>
<td>LU High</td>
<td>604,136</td>
<td>1,130</td>
<td>594,429</td>
<td>1,826</td>
<td>511,886</td>
</tr>
<tr>
<td>Heat</td>
<td>57,986</td>
<td>912</td>
<td>58,912</td>
<td>1,223</td>
<td>58,347</td>
</tr>
</tbody>
</table>
### SLAM Steal Granularity

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQueens Low</td>
<td>80.44</td>
<td>31.17</td>
<td>18.98</td>
<td>10.90</td>
<td>0</td>
<td>372.96</td>
<td>331.99</td>
<td>295.27</td>
<td>145.97</td>
<td>0</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>3,382</td>
<td>2,902</td>
<td>1,868</td>
<td>52.54</td>
<td>0</td>
<td>3,668</td>
<td>3,231</td>
<td>2,164</td>
<td>1,983</td>
<td>0</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>55.87</td>
<td>2.28</td>
<td>0.14</td>
<td>0.05</td>
<td>0</td>
<td>685.80</td>
<td>345.26</td>
<td>166.19</td>
<td>167.50</td>
<td>0</td>
</tr>
<tr>
<td>LU Low</td>
<td>4.57</td>
<td>0.61</td>
<td>0.26</td>
<td>0.20</td>
<td>0</td>
<td>463.02</td>
<td>48.06</td>
<td>21.57</td>
<td>6.58</td>
<td>0</td>
</tr>
<tr>
<td>F7 Low</td>
<td>58.32</td>
<td>23.63</td>
<td>1.00</td>
<td>0.92</td>
<td>0</td>
<td>504.30</td>
<td>296.14</td>
<td>49.59</td>
<td>24.96</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky</td>
<td>0.28</td>
<td>0.05</td>
<td>0.02</td>
<td>0.02</td>
<td>0</td>
<td>209.11</td>
<td>94.69</td>
<td>35.39</td>
<td>40.03</td>
<td>0</td>
</tr>
<tr>
<td>Nbib</td>
<td>63.64</td>
<td>22.24</td>
<td>12.21</td>
<td>5.96</td>
<td>3.12</td>
<td>2,064</td>
<td>2,102</td>
<td>938.99</td>
<td>291.33</td>
<td>181.74</td>
</tr>
<tr>
<td>NQueens High</td>
<td>527.31</td>
<td>189.35</td>
<td>95.11</td>
<td>44.81</td>
<td>23.55</td>
<td>2,083</td>
<td>2,281</td>
<td>2,010</td>
<td>710.78</td>
<td>383.37</td>
</tr>
<tr>
<td>F7 High</td>
<td>137.26</td>
<td>1.38</td>
<td>1.11</td>
<td>1.17</td>
<td>0</td>
<td>972.30</td>
<td>129.85</td>
<td>119.84</td>
<td>41.103</td>
<td>0</td>
</tr>
<tr>
<td>LU High</td>
<td>13.49</td>
<td>1.32</td>
<td>0.39</td>
<td>0.24</td>
<td>0.20</td>
<td>3.35</td>
<td>456.10</td>
<td>456.10</td>
<td>456.10</td>
<td>456.10</td>
</tr>
<tr>
<td>Heat</td>
<td>21.79</td>
<td>10.70</td>
<td>4.51</td>
<td>1.65</td>
<td>1.36</td>
<td>85.26</td>
<td>55.63</td>
<td>26.29</td>
<td>20.87</td>
<td>20.06</td>
</tr>
</tbody>
</table>

Table 5.12: Average, Maximum and Minimum steal granularities running SLAM applications using CMPs with shared Level 2 caches. The values are given in millions of simulation cycles for the cases of Average and Maximum and in simulation cycles in the case of Minimum.

### Cilk Steal Granularity

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQueens Low</td>
<td>44.30</td>
<td>19.11</td>
<td>11.12</td>
<td>7.08</td>
<td>0</td>
<td>466.96</td>
<td>167.63</td>
<td>83.39</td>
<td>57.27</td>
<td>0</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>2,531</td>
<td>2,072</td>
<td>1,516</td>
<td>86.82</td>
<td>0</td>
<td>3,343</td>
<td>2,212</td>
<td>1,017</td>
<td>392.28</td>
<td>0</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>66.91</td>
<td>3.47</td>
<td>0.84</td>
<td>0.71</td>
<td>0</td>
<td>936.99</td>
<td>361.24</td>
<td>256.68</td>
<td>210.20</td>
<td>0</td>
</tr>
<tr>
<td>LU Low</td>
<td>2.48</td>
<td>0.85</td>
<td>0.52</td>
<td>0.91</td>
<td>0</td>
<td>462.72</td>
<td>62.71</td>
<td>26.41</td>
<td>93.41</td>
<td>0</td>
</tr>
<tr>
<td>F7 Low</td>
<td>19.77</td>
<td>10.70</td>
<td>4.59</td>
<td>3.35</td>
<td>0</td>
<td>197.22</td>
<td>213.85</td>
<td>127.09</td>
<td>47.06</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky</td>
<td>0.79</td>
<td>0.59</td>
<td>0.37</td>
<td>0.05</td>
<td>0</td>
<td>245.00</td>
<td>390.80</td>
<td>357.79</td>
<td>78.54</td>
<td>0</td>
</tr>
<tr>
<td>Nbib</td>
<td>206.72</td>
<td>72.09</td>
<td>36.75</td>
<td>25.42</td>
<td>15.86</td>
<td>2,850</td>
<td>1,164</td>
<td>632.36</td>
<td>366.61</td>
<td>193.26</td>
</tr>
<tr>
<td>NQueens High</td>
<td>292.29</td>
<td>115.72</td>
<td>57.04</td>
<td>32.39</td>
<td>20.58</td>
<td>2,863</td>
<td>1,195</td>
<td>915.90</td>
<td>315.89</td>
<td>198.03</td>
</tr>
<tr>
<td>F7 High</td>
<td>53.25</td>
<td>20.15</td>
<td>7.65</td>
<td>4.75</td>
<td>2.93</td>
<td>548.80</td>
<td>578.97</td>
<td>206.92</td>
<td>68.34</td>
<td>205.15</td>
</tr>
<tr>
<td>LU High</td>
<td>8.99</td>
<td>3.08</td>
<td>1.36</td>
<td>1.07</td>
<td>0.23</td>
<td>3,568</td>
<td>506.09</td>
<td>184.27</td>
<td>67.63</td>
<td>522.92</td>
</tr>
<tr>
<td>Heat</td>
<td>8.54</td>
<td>4.78</td>
<td>2.20</td>
<td>1.21</td>
<td>3.69</td>
<td>88.88</td>
<td>56.97</td>
<td>264.44</td>
<td>78.71</td>
<td>384.56</td>
</tr>
</tbody>
</table>

Table 5.13: Average, Maximum and Minimum steal granularities running Cilk applications using CMPs with shared Level 2 caches. The values are given in millions of simulation cycles for the cases of Average and Maximum and in simulation cycles in the case of Minimum.
Figure 5.11: SLAM relative speedup when running in CMPs with private Level 2 caches.

Figure 5.12: SLAM relative speedup when running in CMPs with shared Level 2 caches.
5.8.2 Execution Time of Private VS Shared L2 Caches

The evaluation of SLAM using CMPs with private and shared level 2 caches has shown that in general, using shared level 2 caches for the applications evaluated provides better speedups, which can be observed in Figures 5.11 and 5.12. The execution time however, shows a different situation. Figure 5.13 shows the execution time breakdowns for the divide-and-conquer applications evaluated; Figure 5.14 shows the same information for the loop-level parallel applications. The graphs show both, CMPs with private and shared level 2 caches. The times are normalized to the execution time of the CMPs with private level 2 caches for each case. For the cases that show 64 cores, the values are normalized to the execution times of the corresponding 32-core configuration using private level 2 caches.

It can be seen that Merge Sort, Quick Sort, FFT Low, FFT High and Matmult had longer execution times when using shared level 2 caches for the 32-core configurations. FFT High also took longer even when using 64 cores and shared level 2 caches. Matmult also showed longer execution times when using shared level 2 caches, being from 7.42 to 12.63 times slower relative to the corresponding private level 2 caches configurations.

The cases of Nfib, NQueens Low and High, LU Low and High, Heat and Integration, showed shorter execution times when using shared level 2 caches and 32 cores.

The reason for this is that in this last group of applications, the computations are performed on data that either is generated locally (as is the case of Nfib, NQueens and Integration), or is divided in blocks (as is the case of LU and Heat). This fact favours data locality, minimizing in this way cache latencies.

On the other hand, in applications like Merge Sort, Quick Sort, FFT and Matmult, the computation is performed all over a big array of elements which is accessed by all the cores in the system. This causes the migration based NUCA protocol [43] to make adjustments constantly, trying to keep data in a good location inside the group of level 2 caches. These adjustments eventually lead to poor data locality, which in turn leads to longer execution times.
Figure 5.13: SLAM execution time breakdowns for the divide-and-conquer parallel applications. The values are normalized to the execution times using private Level 2 caches. In the case of 64 cores, the values are normalized to the execution times using 32 cores with private L2 caches.
Figure 5.14: SLAM execution time breakdowns for the loop-level parallel applications. The values are normalized to the execution times using private Level 2 caches. In the case of 64 cores, the values are normalized to the execution times using 32 cores with private L2 caches.

5.8.3 Task Granularity of SLAM Applications

Table 5.14 shows the average task sizes and the parallel phase lengths for the applications evaluated using CMPs with private level 2 caches. It can be seen that there is a tendency in which the task size is increased as the number of cores is bigger, which in turn has a direct impact in the corresponding parallel phase length. This is because in general, the more cores there are in the system, the poorer the data locality.

Similarly Table 5.15 shows the average task sizes and the parallel phase lengths of the applications evaluated using CMPs with shared level 2 caches. It can be observed that the tendency mentioned above is followed by all the applications, with the very clear exception of Heat. In this application the task size is reduced when using 16 and 32 cores. This is because having more cores in the system allowed a larger proportion of the data involved in the computation to be cached. In this way the task sizes corresponding to 16 and 32 cores were smaller than the 8-core ones. However the task size increased again when going from 32 to 64 cores. It was due to the data locality using 64 cores being slightly poorer than the 32-core one.

5.8.4 Level 1 Stack Cache Read and Write Ratios

Table 5.16 shows the values corresponding to the read and write Level 1 Stack Cache Hit ratios when using CMPs with private level 2 caches. It can be observed that only Merge Sort, Quick Sort and Heat did not have 100% write ratios for all
Table 5.14: Task granularities and parallel phase lengths running SLAM applications using CMPs with private Level 2 caches. The granularities are given in cycles whilst the parallel phase lengths are given in millions of cycles.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQueens Low</td>
<td>1.296</td>
<td>1.345</td>
<td>1.540</td>
<td>1.479</td>
<td>2.627</td>
<td>1.545</td>
<td>1.545</td>
<td>5.204</td>
<td>2.636</td>
<td>1.900</td>
<td>1.628</td>
<td>1.024</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>512</td>
<td>900</td>
<td>1.983</td>
<td>3.621</td>
<td>1.642</td>
<td>1.467</td>
<td>1.261</td>
<td>1.028</td>
<td>1.028</td>
<td>1.028</td>
<td>1.028</td>
<td>1.028</td>
</tr>
<tr>
<td>LU Low</td>
<td>39.320</td>
<td>39.529</td>
<td>50.574</td>
<td>63.427</td>
<td>87.066</td>
<td>1.162</td>
<td>1.264</td>
<td>987.2</td>
<td>350.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FFT Low</td>
<td>21.586</td>
<td>44.524</td>
<td>44.813</td>
<td>39.251</td>
<td>65.803</td>
<td>741.4</td>
<td>378.2</td>
<td>169.3</td>
<td>153.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FFT High</td>
<td>30.429</td>
<td>56.886</td>
<td>62.706</td>
<td>51.810</td>
<td>70.362</td>
<td>1.762</td>
<td>899.3</td>
<td>417.0</td>
<td>298.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky</td>
<td>550</td>
<td>724</td>
<td>1.154</td>
<td>1.960</td>
<td>3.677</td>
<td>1.351</td>
<td>1.082</td>
<td>948.7</td>
<td>918.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Heat</td>
<td>242.172</td>
<td>374.673</td>
<td>312.269</td>
<td>355.537</td>
<td>456.719</td>
<td>6.555</td>
<td>3.075</td>
<td>1.780</td>
<td>1.705</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.15: Task granularities and parallel phase lengths running SLAM applications using CMPs with shared Level 2 caches. The granularities are given in cycles whilst the parallel phase lengths are given in millions of cycles.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQueens Low</td>
<td>1.240</td>
<td>1.283</td>
<td>1.312</td>
<td>1.375</td>
<td>2.275</td>
<td>0</td>
<td>1.527</td>
<td>780.0</td>
<td>408.6</td>
<td>325.7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>674</td>
<td>1.192</td>
<td>2.099</td>
<td>2.986</td>
<td>7.148</td>
<td>0</td>
<td>4.232</td>
<td>4,411</td>
<td>3,660</td>
<td>4,341</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>620</td>
<td>1.160</td>
<td>2.755</td>
<td>2.436</td>
<td>4.370</td>
<td>0</td>
<td>2.050</td>
<td>2,085</td>
<td>1,645</td>
<td>1,568</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LU Low</td>
<td>36.342</td>
<td>43.155</td>
<td>45.748</td>
<td>49.148</td>
<td>70.673</td>
<td>0</td>
<td>1,268</td>
<td>680.9</td>
<td>374.7</td>
<td>284.9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FFT Low</td>
<td>44.834</td>
<td>76.408</td>
<td>95.580</td>
<td>77.738</td>
<td>108.577</td>
<td>0</td>
<td>1,264</td>
<td>796.9</td>
<td>337.9</td>
<td>239.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky</td>
<td>614</td>
<td>7.53</td>
<td>896</td>
<td>1.355</td>
<td>3.155</td>
<td>0</td>
<td>1.340</td>
<td>839.5</td>
<td>763.2</td>
<td>763.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nfib</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>1.745</td>
<td>883.0</td>
<td>449.3</td>
<td>246.0</td>
<td>140.2</td>
<td>0</td>
</tr>
<tr>
<td>NQueens High</td>
<td>1.380</td>
<td>1.419</td>
<td>1.561</td>
<td>1.487</td>
<td>2.090</td>
<td>2.519</td>
<td>9.576</td>
<td>5.231</td>
<td>2.508</td>
<td>1.709</td>
<td>1.023</td>
<td>0</td>
</tr>
<tr>
<td>FFT High</td>
<td>55.736</td>
<td>92.855</td>
<td>0</td>
<td>102.482</td>
<td>132.232</td>
<td>132.232</td>
<td>2.878</td>
<td>0</td>
<td>931.7</td>
<td>635.9</td>
<td>420.6</td>
<td>0</td>
</tr>
<tr>
<td>LU High</td>
<td>37.175</td>
<td>43.141</td>
<td>48.760</td>
<td>49.287</td>
<td>64.923</td>
<td>83.602</td>
<td>9.973</td>
<td>5.623</td>
<td>2.875</td>
<td>1.936</td>
<td>1.291</td>
<td>0</td>
</tr>
<tr>
<td>Heat</td>
<td>288.224</td>
<td>388.134</td>
<td>581.656</td>
<td>501.574</td>
<td>300.256</td>
<td>357.878</td>
<td>7.609</td>
<td>5.721</td>
<td>2.486</td>
<td>764.2</td>
<td>434.3</td>
<td>0</td>
</tr>
</tbody>
</table>
the configurations tested. In all the other cases, the read and write ratios were 100%. For the cases where L1 Stack Cache misses were presented, these were treated as a normal L1 cache miss, with the corresponding timing operations involved.

Similarly Table 5.17 shows the read and write Level 1 Stack Cache Hit ratios corresponding to SLAM applications running on CMPs with shared level 2 caches. In this case, only Quick Sort, FFT High and Heat showed write ratios below 100% for some configurations. The same as the case of CMPs with private L2 caches, these Level 1 Stack Cache misses were processed as normal L1 caches misses with their corresponding timing.

All these values correspond to the SLAM configuration described in Table 5.4

5.9 Summary

This chapter presented the evaluation of the proposed SLAM implementation performed in this work. It described the simulation methodology as well as the simulation tools used for carrying out such evaluation. A description of the benchmarks used was provided with the corresponding inputs used in each experiment.

The results of the performance evaluation of the proposed SLAM implementation were presented. This evaluation included a performance comparison between SLAM and Cilk [12] in terms of execution time. The results included the single thread evaluation of SLAM and Cilk versus a plain C implementation of each of the applications used. These results showed that SLAM overhead was small when executing an application with a single thread.

Besides the execution time, the parallel evaluation also included a comparison of the runtime systems of SLAM and Cilk. This comparison involved the measurement of the stealing time and the steal granularity produced in each experiment. It also involved the measurement of the values of the Minimum, Maximum and Average Steal Granularities, as well as the Steal Granularity Distribution during the execution of each application. These measurements were useful to evaluate the load balancing efficiency achieved by SLAM and Cilk. The results were presented in two different scenarios: when using CMPs with private level 2 caches and, when using CMPs with shared level 2 caches. The experiments showed that SLAM was faster than Cilk in all the applications tested when running with the maximum number of cores and using shared level 2 caches. A similar result was
### Table 5.16: Read (R) and Write (W) Level 1 cache hit ratios when running SLAM applications using CMPs with private Level 2 caches.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfib</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>NQueens Low</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>NQueens High</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100/99.96</td>
<td>100</td>
</tr>
<tr>
<td>LU Low</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>FFT Low</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>FFT High</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cholesky</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Heat</td>
<td>99.99</td>
<td>100</td>
<td>99.99</td>
<td>100</td>
<td>99.99</td>
<td>100</td>
</tr>
</tbody>
</table>

### Table 5.17: Read (R) and Write (W) Level 1 cache hit ratios when running SLAM applications using CMPs with shared Level 2 caches.

<table>
<thead>
<tr>
<th>App/CPUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NQueens Low</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100/99.97</td>
<td>100</td>
<td>99.82</td>
<td>0</td>
</tr>
<tr>
<td>LU Low</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>FFT Low</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Nfib</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>NQueens High</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>FFT High</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100/99.94</td>
<td>100</td>
<td>99.92</td>
<td>100</td>
</tr>
<tr>
<td>LU High</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99.92</td>
</tr>
<tr>
<td>Heat</td>
<td>100</td>
<td>99.99</td>
<td>100</td>
<td>99.99</td>
<td>100</td>
<td>99.99</td>
<td>100</td>
</tr>
</tbody>
</table>
obtained when using private level 2 caches, with the exception of \textit{NQueens Low} and \textit{NQueens High}, for which Cilk showed better performance.

The relative speedups achieved by SLAM for each application were also presented. These results again were shown for the cases of the CMPs with private and the ones with shared level 2 caches. The average task size for each application was also covered, as well as the corresponding parallel phase lengths. These numbers showed how the task size varied with the number of cores used in the simulations, which in turn, produced a direct impact on the parallel phase lengths of each application.

Finally an execution time comparison between SLAM systems with private and shared level 2 caches was presented. This comparison included the breakdown of the SLAM execution times showing running, enqueueing, dequeueing and stealing times. These results showed that, when running applications that favor data locality, shared level 2 cache configurations were faster than their private level 2 cache counterparts. For the applications that do not favor data locality, the private level 2 cache configurations resulted to be faster.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

The increasing number of cores built in Chip Multiprocessors has led to the need for programming models that allow the exploitation of parallelism with a reasonable level of complexity. On the other hand, having a high number of cores executing a parallel application can lead to very fine grain tasks. The migration cost of these tasks in a shared memory multicore environment can downgrade performance significantly due to low computation/communication ratios.

This thesis addresses these problems presenting an implementation of SLAM [81] designed to work on Shared Memory Tiled Chip Multiprocessors. The proposed system includes hardware support that allows the exploitation of fine and coarse grain parallelism. This is done using a stack based computation model based on declarative programming principles coupled with a custom communication mechanism. These elements together provide an efficient dynamic load balancing system for shared memory Chip Multiprocessors.

The experiments performed showed that the proposed SLAM implementation outperforms Cilk [29] in terms of execution time and scalability, as can be seen on the results presented in the previous chapter in sections 5.6.2 and 5.7.2. The custom hardware support allows the proposed system to exploit parallelism in situations in which Cilk was not able to do so. This is thanks to the implemented load balancing system that uses the processor interconnect instead of the shared memory to achieve task migration related operations.

The results presented in sections 5.6.3 and 5.7.3 show that SLAM’s load balancing system keep SLAM workers stealing from other workers a low fraction
of the total execution time. The results presented in sections 5.6.4 and 5.7.4 show that SLAM produced steal granularities larger or equal to 10K cycles when executing parallel applications. The low stealing time and the steal granularities spent/produced by SLAM workers are the main components that contribute to make SLAM’s load balancing system an efficient dynamic scheduler. Another factor that contributed to improve the proposed system’s performance, was the low overhead associated with enqueueing/dequeueing tasks to/from the local pool of ready-to-execute tasks. This low overhead is reflected in the results presented in sections 5.6.1 and 5.7.1, and also can be observed in the results presented in section 5.8.2.

The contributions of this thesis are:

- A dynamic load balancing system based on SLAM adapted to work in tiled shared memory Chip Multiprocessors. This system provides a very simple stack based mechanism for evaluating parallel applications. It performs computation, synchronization and communication activities using the stack, which greatly simplifies its implementation.

- Appropriate hardware support for increasing the efficiency of the load balancing system, making it capable of exploiting fine grain parallelism when required, but being also able to work with coarse grain applications. It includes a custom communication mechanism that performs task migration using the processor interconnect. This allows the system to have a better scalability since it does not use the shared memory as the core for achieving load balancing related operations.

- A fully cache coherent evaluation of the proposed SLAM implementation using Tiled Shared Memory Chip Multiprocessors ranging from 4 to 64 cores. The implementation was evaluated using CMPs with private Level 2 caches and CMPs with shared Level 2 caches. Our experiments showed that our implementation of SLAM clearly outperforms Cilk for the group of applications tested.

- A full system simulation platform for SLAM. This platform includes operating system support, directory-based cache coherent memory system, point-to-point mesh processor interconnection network and SLAM hardware acceleration support.
• A SLAM emulator. This is an emulator capable of running multithreaded SLAM applications on real multiprocessor machines (without SLAM hardware support). It is implemented using the Pthreads library [15] and has been tested on X86 and AMD64 machines running Linux, and on a Sun UltraSPARC T2 machine running Solaris. This tool allows a faster development of SLAM applications, since these can be tested on real hardware at real speeds in order to check for correctness.

6.2 Future Work

In this thesis, a SLAM implementation for Shared Memory Tiled Chip Multiprocessors has been proposed and evaluated. The results of performance and scalability of the mentioned system are encouraging. Something that remains to be done is a more extensive evaluation using different and more diverse parallel applications. Another aspect to explore would be the power consumption of the overall system. These studies would give a clearer view of the potential of the proposed system regarding its capabilities.

It would be interesting to look at the behavior of SLAM in a multicore system with simultaneous multithreading capabilities such as Ultrasparc T2 [63], with the purpose of experimenting with different stealing policies implemented in hardware. For example, a very obvious experiment would be to try to steal first from a SLAM worker running on a thread located in the same core as the thief. Another one could be to steal for all the starving SLAM workers running on the same core when stealing from a different core. This would allow the study of different ways of improving data locality when running parallel applications in this kind of environment.

Another interesting experiment would be to investigate how to build a more flexible SLAM runtime system. This could be one in which the custom hardware would provide information to a more flexible software runtime system in order to achieve better dynamic load balancing. A system with similar features is presented in [22], however SLAM’s simple and unified stack-based execution/synchronization mechanism would produce a different system for investigation.

With the growth of multicore systems, novel mechanisms for exploiting parallelism have appeared on the scene. One of these mechanisms is Transactional Memory (TM) [38]. TM has been proposed using purely hardware schemes [31,
62, 7], purely software schemes [45, 37, 33] and hybrid hardware/software sys-
tems [21, 59]. Another possible direction to follow would be to integrate a HTM
or HybridTM system with SLAM. This could result in a shared memory multi-
core system with more flexible hardware for exploiting parallelism, which would
not be restricted to a purely functional form. A similar idea, but implemented in
software is presented in [34].
Bibliography


