PARALLELIZATION AND PERFORMANCE-TUNING:

AUTOMATING TWO ESSENTIAL TECHNIQUES IN THE MULTICORE ERA

A Thesis

Submitted to the Faculty

of

Purdue University

by

Chirag Uday Dave

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2010

Purdue University

West Lafayette, Indiana
To my parents, Meena and Uday, and my brother, Yash.
I would first like to thank my advisor, Prof. Rudi Eigenmann, for his advice, support and understanding over the course of my graduate studies at Purdue. He has always lent an attentive ear to every issue I have come across, has provided much needed help in getting through various difficult situations and it has been a pleasure to work with him. I would like to thank Prof. Sam Midkiff for all his support and advice related to research as well as succeeding at graduate school. I would also like to thank my committee member, Prof. Zhiyuan Li for his support.

The Paramount Research Group at Purdue comprises of great people and I shall take with me good memories of my times with the group. I’d like to thank Seyong Lee, Okwan Kwon, Fahed Jubair, Aurangzeb and Dheya Mustafa for their friendship and support. Special thanks to Hansang Bae, Sang-Ik Lee and Seung-Jai Min for their highly valuable guidance and support in work related to the Cetus Compiler Project.

I also appreciate the mentoring and guidance of Guojing Cong at the IBM T.J. Watson Research Center during my internship in Summer 2008. My time there was a motivating factor in driving some parts of my research.

I could not have succeeded during my graduate school career without the love and support of my family. They have been my pillars of strength during these years and I thank them from the bottom of my heart. Also, thanks to all my friends at Purdue University with whom I have shared great experiences and will cherish these memories throughout life.

Finally, I must thank God for his blessings and for having helped me choose the right path in life and follow it through successfully. I am truly grateful.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xi</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Automatic Parallelization</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Automatic Performance Tuning</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>6</td>
</tr>
<tr>
<td>2 BACKGROUND AND MOTIVATION</td>
<td>7</td>
</tr>
<tr>
<td>3 CETUS: AUTOMATIC PARALLELIZATION</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Infrastructure Overview</td>
<td>12</td>
</tr>
<tr>
<td>3.1.1 Cetus High-level Intermediate Representation (HIR)</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Data-dependence analysis</td>
<td>14</td>
</tr>
<tr>
<td>3.2.1 Eligibility Identification</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2 Symbolic Range Analysis Support</td>
<td>20</td>
</tr>
<tr>
<td>3.2.3 Information Collection</td>
<td>21</td>
</tr>
<tr>
<td>3.2.4 Dependence Test</td>
<td>23</td>
</tr>
<tr>
<td>3.2.5 Data Dependence Graph (DDG)</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Alias Analysis</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1 Conservative alias analysis with manual input</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Points-to analysis</td>
<td>29</td>
</tr>
<tr>
<td>3.4.1 Iterative work-list based control flow analysis</td>
<td>33</td>
</tr>
<tr>
<td>3.4.2 Points-to Domain and Points-to Rel</td>
<td>34</td>
</tr>
<tr>
<td>3.4.3 Pointer Symbols: Stack allocated and heap allocated variables</td>
<td>35</td>
</tr>
<tr>
<td>3.4.4 Pointer assignment</td>
<td>36</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Null Domain and Universe Domain</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Function Call handling: Interprocedural Overview</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Normalization</td>
</tr>
<tr>
<td>3.4.8</td>
<td>Applications</td>
</tr>
<tr>
<td>3.5</td>
<td>Loop Parallelization</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Interface to DDG</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Handling scalar dependences</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Direction Vectors and Serializing Outer Loops</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Cetus Annotations</td>
</tr>
<tr>
<td>4</td>
<td>AUTOTUNING OF PARALLEL PROGRAMS</td>
</tr>
<tr>
<td>4.1</td>
<td>Cetune: Automatic Tuning Framework</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Methodology</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Tuning Algorithm</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Tuner implementation in Cetus</td>
</tr>
<tr>
<td>4.1.4</td>
<td>System Workflow</td>
</tr>
<tr>
<td>4.2</td>
<td>Evaluation</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Tuning Auto-Parallelized Code</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Tuning Additional Parallelism in Hand-Parallelized Code</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Tuning time</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Intel's ICC Auto-Parallelizer</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Dynamic Tuning Considerations</td>
</tr>
<tr>
<td>5</td>
<td>CONCLUSIONS</td>
</tr>
<tr>
<td></td>
<td>LIST OF REFERENCES</td>
</tr>
</tbody>
</table>
4.1 Number of loops automatically parallelized by Cetus (only the loop that is parallel and not enclosed by parallel loops is considered) and those already present in the input are shown in this table for a subset of the NAS Parallel and SPEC OMP2001 benchmarks. The number of Cetus-parallelized loops forms the search space for our tuning approach. Parallel loops that are nested inside outer loops can produce significant overheads due to repeatedly spawning and joining parallel threads. Selecting the right loops out of all the parallelized loops is crucial to obtaining better performance.
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Intel CPU trends over the last three decades.</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>The Cetus Intermediate Representation implemented using Java class hierarchy. The horizontal arrow relationships represent object-oriented hierarchical IR features while the vertical arrow relationships define the tree structure of the IR i.e. the parent-child relationships.</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>High-level data dependence analysis algorithm. Collects eligible loop-nests and provides the necessary loop information for testing pairs of array accesses. A data dependence graph is provided as output.</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Example code showing eligibility of loops. The loop containing a function call renders the outermost loop ineligible as well. The third and fourth loops at nesting levels 2 and 3 are eligible. Symbolic bounds such as ‘ub’ can be handled, provided the initialization, condition and increment expressions are canonical.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>The Data dependence wrapper traverses the IR and collects references to relevant IR information for local storage purposes. This information is then passed to the dependence test.</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Data dependence driver algorithm and wrapper for a <em>single loop-nest</em>. Traverses list of array accesses and builds the loop dependence graph.</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Dependence test wrapper that handles actual partitioning and traversal of the pairs of subscripts that are being tested. Handles low level direction-vector traversal and calls the actual dependence test e.g. Banerjee test</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>High-level intraprocedural points-to algorithm. Uses flow-sensitive approach by building a control flow graph and returns a map from statement to points-to domain for the entire procedure.</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>Iterative traversal of the reverse-post ordered work-list. This traversal is used to analyze the control flow graph for points-to updates. The merge operation is defined in [31].</td>
<td>31</td>
</tr>
<tr>
<td>3.9</td>
<td>Handling of assignment statements and function calls during data-flow analysis. Through normalization, most statements can be converted to conform to one of these two constructs. See Section 3.4.7 for normalization details.</td>
<td>32</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.10 Simple example showing PointsToRel and PointsToDomain, printed by Cetus before their corresponding program points.</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>3.11 processAssignment handles the crucial implementation for the data-flow transfer function. LHS and RHS locations are computed, and gen, kill, union(∪) and diff(−) operations are performed during the assignment update.</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3.12 Example code showing conservative points-to update via the UniverseDomain. Such cases mostly arise due to conservative assumptions in other situations such as non-normalized or unknown expressions or unhandled function calls.</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>3.13 Points-to information is a superset of the NullDomain. However, a union with the UniverseDomain will always result in universe, thus representing the most conservative assumption of points-to information at that program point as well as for the subsequent program.</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>3.14 Expression normalization examples for points-to analysis. These internal and temporary transformations help simplify the analyzer and help handle certain corner cases.</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>3.15 This figure shows the difference in the number of loops parallelized by Cetus under different alias assumptions or analyses. all-alias is where no assumptions are made about function arguments and conservative alias information is used. argument-noalias-global allows manual input of information showing that arguments are not aliased to each other or to global variables. Accurate interprocedural points-to analysis, alias-advanced-interproc gives close to the best possible alias results and hence, close to the FORTRAN assumption results on 9 out of 10 benchmarks.</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>3.16 Example code showing importance of points to analysis for techniques such as constant propagation. The if condition in this example would evaluate to false, but this would be impossible to determine without points-to information. Optimizations such as constant propagation and hence, dead-code elimination can thus benefit.</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>3.17 High-level loop parallelization algorithm. Uses the data dependence graph and privatization and reduction information. Direction vector traversal and elimination of covered dependences detailed in [26].</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>
4.1 High-level overview of the offline empirical tuning system, Cetune. A search algorithm navigates through the space of possible optimization variants. It makes a decision on the next suitable optimization variant that the system should try. This variant is handed to the version generator in order to be compiled. Performance of each version is measured using execution time, the result of which feeds back into the search-space navigation algorithm, which determines the next suitable variant, until a convergence criterion is satisfied.

4.2 Parallelized source code (a) and the different variants of loop parallelization possible (b). (a) shows the output obtained from Cetus after loop parallelization. The tuner uses the number of parallel loops as its search space and then navigates this space using Combined Elimination to obtain the final binary vector configuration. Cetus’ output pass is used to parse the commandline vector shown in (b) and generate OpenMP directives corresponding to the input. The entire process is automatic.

4.3 This figure shows the complete training phase of our tuning implementation. Starting with auto-parallelization, the tuner starts iteration from a specific start configuration (combination of parallel loops), moving to successive configuration points on each iteration. A final configuration is derived from the trained input and is applied for evaluation to other reference data sets. (RIP: Relative Improvement Percentage)

4.4 Cetune: Tuning results for BT, CG, EP, FT and LU benchmarks. Training performed with W data set for NAS Parallel suite and train data set for SPEC OMP2001 suite. CetusParallel shows performance for compiler-parallelized code. Cetuned tunes only Cetus-parallelized loops and aims to perform close to equal or better than Serial.

4.5 Cetune: Tuning results for MG, SP, equake and art benchmarks. Training performed with W data set for NAS Parallel suite and train data set for SPEC OMP2001 suite. CetusParallel shows performance for compiler-parallelized code. Cetuned tunes only Cetus-parallelized loops and aims to perform close to equal or better than Serial.

4.6 Tuning results for FT and art benchmarks by using a different start point for the search space. Training performed with A data set for FT and train data set for art. CetusParallel shows performance for compiler-parallelized code. Cetuned tunes only Cetus-parallelized loops and performs close to equal or better than Serial. HandParallel shows performance of manual parallelization.
4.7 **Hand+Cetune**: Tuning results for BT, CG, EP, FT and LU benchmarks. Training performed with W data set for NAS Parallel suite and *train* data set for SPEC OMP2001 suite. *Hand+Cetuned* tunes parallel loops that exist in addition to hand-parallel loops *HandParallel*.  

4.8 **Hand+Cetune**: Tuning results for MG, SP, equake and art benchmarks. Training performed with W data set for NAS Parallel suite and *train* data set for SPEC OMP2001 suite. *Hand+Cetuned* tunes parallel loops that exist in addition to hand-parallel loops *HandParallel*.  

---

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>67</td>
</tr>
<tr>
<td>4.8</td>
<td>68</td>
</tr>
</tbody>
</table>
ABSTRACT


In today’s multicore era, parallelization of serial code is essential in order to exploit the architectures’ performance potential. Parallelization, especially of legacy code, however, proves to be a challenge as manual efforts must either be directed towards algorithmic modifications or towards analysis of computationally intensive sections of code for the best possible parallel performance, both of which are difficult and time-consuming. Automatic parallelization uses sophisticated compile-time techniques in order to identify parallelism in serial programs, thus reducing the burden on the program developer. This work considers the implementation of important parallelization techniques such as Data dependence analysis and advanced Points-to and Alias analysis in a source-to-source parallelizing compiler, Cetus. Auto-parallelization results are provided across a set of benchmarks from the NAS Parallel and SPEC OMPM2001 suites.

A key difficulty in using automatic parallelization, however, is that optimizing compilers are generally unable to estimate the performance of an application or even a program section at compile-time, and so the task of performance improvement invariably rests with the developer. Automatic tuning uses static analysis and runtime performance metrics to determine the best possible compile-time approach for optimal application performance. This work describes an offline tuning approach that uses Cetus with an additional tuning framework to tune parallel application performance. An existing, generic tuning algorithm called Combined Elimination is used to study the effect of serializing parallelizable loops based on measured whole program
execution time. The outcome is a combination of parallel loops that ensures to equal or improve performance over the original program. The results from the autotuning approach are compared against hand-parallelized C benchmarks from the SPEC OMP2001 and NAS Parallel suites. The auto-parallelized and auto-tuned versions are close to serial performance or better than serial in most cases and always outperform state-of-the-art parallelizers such as Intel’s ICC. Additional parallelization techniques and more extraction of beneficial parallelism can help improve the tuning results further.
1. INTRODUCTION

Over the last few decades, more computational power and faster clock speeds have enabled research into broader areas and this has consequently led to the development of applications that require even greater computational capabilities. Scientific and high-performance computing applications have been the driving forces behind advances in the computing industry; while the industrial and personal computing areas have continued to reap the benefits of these advances.

As predicted by Moore’s Law, the number of transistors on integrated chip sets has doubled every two years, thus allowing the semiconductor industry to satisfy an ever-increasing demand for more computational power. Single processor clock speeds have increased with increasing numbers of transistors, and serial applications have benefited mostly from the subsequent computational power. It is important to note that in such an environment, programmers have solely focused on optimization, performance tuning and debugging in the context of serial programming. Software performance has followed the trends in hardware with the help of strong compiler support and hence without portability concerns. However, in recent years, the hardware computing industry has steered down a different path. Power varies roughly as the cube of the increase in frequency or clocking speed. Thus, doubling the number of transistors according to Moore’s Law as well as clocking the hardware at high frequencies has led to a tipping point for power consumption. Single core processors have reached hot-plate levels of heating, clearly inhibiting further increases in clock speeds, as illustrated by the growth curves in Figure 1.1. In an effort to meet further demands for computational power, the computing industry has turned towards multicore processors as a viable option. Multicore processors can be considered to be a collection of simple processor cores that share onchip resources. More computational
Fig. 1.1. Intel CPU trends over the last three decades.

demand can be met by adding more cores at lower clock speeds, thus providing a linear increase in power consumption.

Shared memory as well as distributed memory multiprocessor systems have been a standard in high-performance computing over the last two decades. The scientific community has developed software programming models for parallel machines in an effort to extract maximum computational performance. Multicore processors can be considered as shared memory parallel systems. Compared to multiprocessor systems, in which whole processor resources have been duplicated and used in parallel, multicore processors contain cores that are much simpler and slower versions of their predecessors. In order to keep power consumption at a minimum, each core is clocked at much slower rates. This brings forth an alarming concern for the software industry. Executing a serial i.e. single-threaded application on a multicore would result in significant degradation in performance unless the software is able to use
the functionality provided by multiple cores. Moreover, multicores are finding their way into a wide-range of computing systems from embedded platforms to personal computer systems to enterprise computing infrastructures. Software parallelism is, thus, of paramount importance in order to make full use of the computational power of multicores.

As mentioned before, the scientific community has used massively parallel computer systems and large scientific parallel applications over the last two decades. Programming parallel machines brings a different set of challenges in the aforementioned areas of optimization, performance tuning and debugging. Programmers have mostly used manual development techniques such as profiling and bottleneck identification as well as manual program analysis to improve the parallel performance of their applications. With the introduction of multicores and the ubiquity of parallel systems, all applications must attempt to exploit parallelism at some level. All programmers would need to have parallel programming knowledge. Manual tweaking of optimizations and performance tuning are no longer feasible due to the lack of portability of parallel performance across different systems. Legacy code that has relied on single-threaded performance improvements must be parallelized for future performance benefits. In such a scenario, parallel application programming is a highly time consuming, complicated and error-prone task. Automated techniques for parallel programming can provide the much needed support for complex analyses and significant benefits in terms of development time.

1.1 Automatic Parallelization

Developing efficient programs for parallel computers is a time-consuming and difficult process, as outlined above. In addition to the skills required by the programmer to create a parallel application, there are other issues such as portability, correctness and scalability that arise due to the complexity of parallel architectures that are seen
today. Porting an entire application from a specific machine to another might be more difficult than having to reprogram it from scratch.

Different solutions to this problem have been proposed over the last decade, one of which is automatic parallelization. Automatic parallelization is the technique of using a parallelizing compiler or translator to transform sequential programs into equivalent parallel programs [1]. An automatic parallelizer is equipped to use static-time analyses and transformations in order to extract sections of code that can be run in parallel. It can be seen that automatic parallelization can be made to target different levels of parallelism and consequently, different programming platforms and models. A backend compiler, for instance, produces instruction level parallelism through effective register allocation and instruction scheduling during the code-generation phase and thus already provides limited parallelization at the instruction level.

In a similar manner, automatic parallelization can be used to target coarse-grained parallelism using source-to-source translators. Various transformations that are beneficial for memory locality or parallelization but cumbersome to implement manually, can be performed through the use of such translators. Automatic parallelization is ideal for the development of programs in a familiar sequential environment. Parallelism is extracted from these programs without the need of manual intervention, thus reducing the burden on the programmer to a large extent. Legacy code that already exists in sequential form can also be parallelized for execution on parallel systems, without having to recreate the program from scratch. Translators can be provided for translation between different programming platforms e.g. sequential software can be automatically translated into a language that makes the application compatible with Graphics processing units, thus targeting General Purpose Graphics Processing Units (GPGPUs).

Automatic parallelization is largely facilitated through static-time analyses and transformations such as scalar and array privatization, scalar and array reduction variable recognition and also induction variable substitution. As the compiler or translator uses these techniques during compilation, they’re mostly independent of
information in the input data set and of the target architecture. Data dependence
analysis is a memory disambiguation technique at the core of identifying independent
sections of code that can be executed in parallel [2]. The design and implementation
of a well-known data dependence analysis technique is described later in Section 3.
Loop transformations such as interchange, loop tiling and fission or distribution help
in the improvement of spatial and temporal locality [3].

The potential of automatically applying the above analyses and transformations
significantly reduces the burden on the programmer of manually creating a parallel
program or parallelizing a given serial program.

1.2 Automatic Performance Tuning

Performance tuning is the process of optimizing the performance of an application
through the right combination of analyses and transformations. Manual performance
analysis is an iterative and intensive process. The programmer must identify perfor-
mance bottlenecks for the application through profiling tools, and choose the right
transformation for improving the performance in the bottleneck section. Choosing
the right transformation involves tough decisions such as legality and implementa-
tion of other enabling analyses and transformations. Traditionally, the serial application
domain has been studied in depth and manual performance tuning has provided best
results as well as performance portability. In the parallel domain, however, perfor-
mance tuning is significantly harder due to reasons such as varying and complex
architectures, scalability issues and also variability in parallel algorithm performance.

Automatic parallelization uses sophisticated compile-time techniques in order to
identify parallelism in serial programs, thus reducing the burden on the program de-
developer. However, similar sophistication is needed to improve the performance of
hand-parallelized as well as auto-parallelized programs. A key difficulty is that opti-
mizing compilers are generally unable to estimate the performance of an application
or even a program section at compile-time, and so the task of performance improve-
ment invariably rests with the developer. Automatic tuning uses static analysis and runtime performance metrics to determine the best possible compile-time approach for optimal application performance [4].

1.3 Organization

The remainder of the thesis is organized as follows. Section 2 introduces the reader to the topics discussed above in more detail and motivates the importance of the work implemented in this thesis. Section 3 provides a detailed explanation of the parallelization techniques implemented in the Cetus compiler infrastructure as part of this thesis. Section 4 presents an automatic tuning framework and its implementation using Cetus. In Section 5, we provide a summary, draw our conclusions and provide directions for future work.
2. BACKGROUND AND MOTIVATION

The premise of the work described within this thesis is as follows: In the multicore era, automatic parallelization and automatic tuning are required to play significant roles in the parallel program development process, especially with the aim of reducing the burden on the programmer and providing improved parallel program performance.

Parallel execution entails that multiple program instructions be executed simultaneously. Parallelism can be extracted at different levels of program execution: instruction-level parallelism being fine-grained while source-level or algorithmic parallelism being coarse-grained. Single thread processors have relied on instruction level parallelism to achieve concurrency in an effort to decrease program execution time. Simultaneous multithreading (SMT) was a step forward [5]: more parallelism was achieved by having multiple threads supported on a single processor, each with its own stream of execution instructions. This was made possible by faster clock speeds and duplication of execution resources in hardware. However, the computing industry is now focused on reducing the complexity of these single thread processors and on duplication of processor cores in an effort to reduce power consumption. This leads to a paradigm shift in the granularity of parallelism now required from applications in order to effectively make use of the computational resources available. A coarser-grain parallel execution model requires expression of parallelism at a higher level, thus requiring the involvement of the programmer.

Parallel programming introduces a multitude of concerns for the programmer. These can range from thread level issues, thread interaction, shared states across threads, data races and locks [6] to application scalability. Parallel programming models have been used to aid the programmer in expressing or encoding parallelism in the source program. This allows the backend compiler to generate code that is capable of exploiting multiple resources simultaneously. These programming models
were initially more akin to assembly level programming, for example, Pthreads [7], whereby the program developer has to explicitly create and identify program sections that could be run in parallel and also identify data passed into each thread explicitly. This puts the onus of correctness as well as performance on the programmer. OpenMP [8], introduced in the early nineties, is a shared memory programming model with support for C and FORTRAN. In OpenMP, the programmer is required to insert directives and other minor parallel constructs into a serial program. A backend OpenMP compiler implementation must automatically translate, using libraries, the directives into code that can be executed in parallel. The OpenMP backend translation is usually done using Pthreads libraries, and thus OpenMP can be considered a higher abstraction of the Pthreads programming model. As opposed to the shared memory abstraction, the Message-Passing Interface (MPI) programming model supports Single Program Multiple Data (SPMD) execution across a distributed computer system [9]. Thread interaction occurs via explicit data communication across processors. For MPI, the programmer must partition the data across processors and include explicit communication calls inside the source code.

All three programming models discussed above vary in the difficulty of implementation from the programmer’s point of view. However, they all require that the programmer make a detailed manual analysis of the source program and also, take complex decisions regarding code that can be executed in parallel. The inherent understanding of serial execution must be transformed into a parallel view of execution. Such complexity and increasing demands for parallel programming skills spurred research in the early nineties regarding the capabilities of compilers for automatically translating serial programs into programs capable of executing in parallel. Eigenmann et al. [10] provided a detailed analysis of the Perfect benchmarks and the manual techniques required to translate them into equivalent parallel programs. The research also considered the feasibility of incorporating these techniques into compilers in order to achieve automatic parallelization. The Polaris compiler [11] was a vastly successful translator for parallelizing FORTRAN programs and was used to implement the tech-
niques described in that research. KAP, a commercial compiler, and SUIF [12] were other automatic parallelizers of that era to provide strong compile-time analyses and parallelization capabilities at the source level.

In today’s multicore era, parallel programming is gaining ubiquitous importance. The ability to parallelize existing applications and gain the available performance benefits are of paramount importance. The Cetus Compiler Infrastructure [13, 14] was initiated and developed at Purdue University, and is targeted towards supporting strong compile-time analyses and translation capabilities for C programs. Cetus is a source-to-source translator for C programs, implemented in Java. It is one of the few parallelizers to provide source level translation capabilities, which can be invaluable to the parallel development process. Cetus provides a range of parallelization analyses and transformation capabilities, all implemented in about 50,000 lines of code. A strong object-oriented API provides the user with huge development and maintenance benefits. Other notable compiler infrastructures that support source level translation are LLVM [15], ROSE [16], COINS [17] and Pluto [18]. While LLVM and ROSE are established compiler infrastructures, they’re both implemented in C++ and are significantly large, thus leading to a huge learning curve e.g. ROSE is over 500,000 lines of code.

Despite much progress in automatic parallelization over the past two decades and despite the new attention this research area is receiving due to the advent of multicore processors, parallelization is still not the default compilation option of today’s compilers. One reason is that there still exist many programs in which today’s parallelizing compilers cannot find significant parallelism. However, a more important reason is that, even where compilers find parallelism, they cannot guarantee that the parallelism is executed beneficially. Users may experience performance degradation unless they invest substantial time in tuning the parallel program. This thesis describes the concept of a parallelizer that performs such tuning automatically, in an effort to ensure that compiler-parallelized code performs at least close to the performance of the original program or better.
State-of-the-art solutions for finding profitable parallel regions are usually simple compile-time models. The compiler may count the number of statements and iterations of a parallelizable loop, for example, and serialize the loop if these counts are known to be less than a user-defined threshold. This semi-automatic method is used in the Polaris parallelizer [11], and Intel’s ICC compiler, for example. Limited compile-time knowledge of program input data, architecture parameters, and the execution environment render these models inaccurate. Several techniques have provided runtime and interactive systems for finding profitable parallel loops [19–21] and improving parallel loop performance. While these solutions have been shown to be effective, they address only one of a large number of problems with compiler techniques that are limited by insufficient static knowledge. A general automatic tuning solution, which can be applied to many compilation techniques is essential for achieving desired performance gains. The tuning method described in this thesis relates to other approaches that navigate a search space of possible transformation options, picking the best through empirical evaluation [22]. This method contrasts with work that directs the search process through performance models [23]. Model-guided tuning has the potential to reduce the search space of possible compilation variants and thus tends to tune faster, while empirical tuning has the advantage of using the ultimate performance metric, which is execution time. The two methods complement each other and such complementary approaches have been proposed that use run-time profile information to drive optimization tuning at compile time [20,24].

Most of today’s tuning systems apply a chosen compilation variant to the whole program. An important un-met need is to customize compiler techniques on a section-by-section basis. Typical tuning systems choose a set of compilation flags, which are then applied to the entire program, even though some approaches have tuned individual subroutines [25]. Finer tuning would allow many additional techniques and parameters to be made amenable to runtime tuning; for example, the most suitable register allocation method could be selected for each basic block. The tuning
framework described in this thesis allows the selection of serial or parallel execution on a loop-by-loop basis.
3. Cetus: Automatic Parallelization

The Cetus Compiler Infrastructure forms an integral part of the work described in this thesis. This section provides an overview of the compiler and its translation capabilities. It then goes on to discuss three analyses - Data dependence analysis, Points-to/Alias analysis and Loop Parallelization - in detail. These analyses constitute a substantial portion of Cetus’ automatic parallelization capabilities. We also provide an overview of the parallelization results on the SPEC OMP and NAS Parallel benchmark suites.

3.1 Infrastructure Overview

Cetus is a source-to-source compiler-translator for C implemented in Java. The compiler started as a class project in 2003 and has grown to about 50,000 lines of code over the past few years. The compiler is aimed at providing a maintenance-friendly, user-oriented development platform for source-to-source compilation strategies [13, 14]. First, we consider a brief overview of the hierarchical structure of the Cetus High-level Intermediate Representation (HIR), which is essential for a better understanding of the Cetus implementation work described later in the section.

3.1.1 Cetus High-level Intermediate Representation (HIR)

A compiler’s intermediate/internal representation (IR) is the storage format that the input source program is converted into in order to represent information in a more compiler-friendly form. The IR can then be traversed, analyzed and transformed by various compiler passes. Following these operations, the output pass reconverts the IR into the target format, which could be object code in the case of a back-end
compiler or, translated source programs as in the case of Cetus. The Cetus HIR is implemented in the form of a Java class-hierarchy [13,14]. A high-level representation provides a syntactic view of the source program to the pass writer, making it easy to understand, access and transform the input program. For example, the Program class type represents the entire program that may consist of multiple source files. Each source file is represented as a TranslationUnit. Other important HIR object types are Statement, Declaration, and Expression. Classes derived from these base classes represent specific source constructs in the HIR as shown graphically in Figure 3.1.

Fig. 3.1. The Cetus Intermediate Representation implemented using Java class hierarchy. The horizontal arrow relationships represent object-oriented hierarchical IR features while the vertical arrow relationships define the tree structure of the IR i.e. the parent-child relationships.
ExpressionStatement represents a Statement that contains an Expression, and an AssignmentExpression represents an Expression that assigns the value of the right-hand side to the left-hand side. There is complete data abstraction, and pass writers only manipulate the HIR through access functions. Important supporting features of the HIR include the following:

- **Traversable:** Most Cetus HIR objects implement the Traversable interface, which allows for easy iteration over lists of objects and their parent-child relationships.

- **Iterators:** BreadthFirst, DepthFirst, and Flat iterators are built into the functionality to provide easy traversal and search over the program representation.

- **Symboltable:** Cetus’ symbol table functionality provides information about identifiers and data types. Its implementation makes direct use of the information stored in declaration statements in the HIR. This link is provided via the Symbol interface. There is no separate and redundant symbol table storage.

- **Annotations:** Comments, pragmas, directives, and other types of auxiliary information about HIR objects can be stored in Annotation objects. The Annotatable interface provides the ability to attach annotations to Cetus HIR constructs (such as information about an OpenMP directive associated with a FOR statement). Standalone annotations such as comment lines can also be inserted in the form of AnnotationStatements.

- **Printing:** Cetus provides customizable printing functions to allow for flexible output of the HIR classes.

### 3.2 Data-dependence analysis

Data dependence analysis is a memory disambiguation technique that seeks to identify data references accessing the same memory locations during program execution. Specifically, it characterizes dependencies between those references [26, 27].
Dependencies can exist between scalar as well as array variables, other aggregate structures being a combination of these variables. Array data dependence analysis involves the process of mathematically analyzing array subscripts to disprove that two subscript computations access the same elements of an array. In source level loops, these array subscripts usually appear as functions of the loop index variable, creating dependencies on, possibly the same or on distinct, statements between different iterations of the loop. Data dependence tests try to find integer solutions to systems of equations introduced by the subscript functions, defined under loop iteration and direction-vector constraints.

A few dependence-related concepts are defined below and serve to clarify the implementation described in the following sections. More details about these concepts are provided in [27].

- **Dependence**: A dependence arises due to the flow of a value between the read and write of a memory location during program execution. The read/write point in the program where the dependence originates is the *source* data reference and the next read/write point in the program to use or overwrite that value is the *sink* reference. This flow of values from source to sink can be characterized further, as described below.

- **True dependence**: A true dependence exists if the sink of the dependence *reads* a value that is *written* to by the source in the immediately preceding write reference to the data. This is also referred to as a *flow dependence* and its existence in the program cannot be eliminated.

- **Anti dependence**: An anti dependence exists if the sink of the dependence *writes* a value to the location that was *read* by the source in the immediately preceding reference to that location. An anti dependence is created due to the re-use of memory locations and can be eliminated by creating a new location for the latter write.
- **Output dependence**: An output dependence exists if the sink *re-writes* to a location that was *written* to in the immediately preceding write reference to that location, which is the source of the dependence. An output dependence can be eliminated in the same way as an anti dependence.

- **Input dependence**: An input dependence exists if the sink *re-reads* a value that was *read* in the immediately preceding reference to that location, which is the source of the dependence. An input dependence does not affect program execution order, and is not relevant to the discussions in this thesis. All relevant dependences must contain at least one write reference.

- **Dependence distance**: Dependencies between source and sink array references occurring inside of loops are characterized in terms of the loop index variable. The dependence *distance* describes the difference in numerical value of the loop iterations across which dependencies exist i.e. how far apart in loop iterations the references to the array location are.

- **Dependence direction**: In many cases, it is easier to identify the existence of a dependence and the general *direction* for the flow of dependence values. For a given dependence across loop iterations, the direction specifies whether the value of the loop index variable for the source reference is less than, equal to or greater than its value for the sink reference. For a multidimensional array subscript enclosed within a multiply-nested loop, a *direction vector* serves the purpose of describing the direction of the dependence with respect to each common loop enclosing the source and the sink references. For example, for a triply-nested loop, a valid direction vector can be represented using the direction values as -1,0,1 or using directional characters as <,=,>.

- **Loop carried dependence**: From the definition of dependence direction, we can infer that the direction must be less than or greater than in order for the dependence to flow across loop iterations. Such a dependence is called
loop-carried and prevents all the iterations of the loop from being executed simultaneously i.e. in parallel.

Cetus uses an array data dependence analyzer during the process of automatic parallelization. The dependence analysis framework is structured into stages: eligibility identification, the information-collection wrapper, the testing framework and the output data dependence graph. Figure 3.2 shows the high-level algorithm that describes the outer-most phases of the analyzer.

3.2.1 Eligibility Identification

This forms the first step of the wrapper around the dependence-testing module. As discussed in the description of data dependence analysis, loops are the primary targets of array dependence analysis. However, the analysis may not always be applicable to a certain loop or loop nest at static time due to a number of reasons. Symbolic loop iteration bounds, for instance, prevent the mathematical dependence module from inferring anything about dependences over the loop iteration space. Also, normalization of the loop step might be required for tests that fail to handle negative loop steps or loop steps not equal to 1. A variable loop step also hinders dependence analysis, as the test cannot make any assumptions about the access pattern for the loop from iteration to iteration. In order to accommodate for these situations, the wrapper traverses the Cetus HIR and determines outermost eligible loops, filtering out the ineligible ones. The requirement for an outermost eligible loop is described by the example in Figure 3.3. Starting with the innermost loops, the outermost-nest-level of eligibility is collected for every nest. It is important to increase eligibility as much as possible, so as to increase the scope of the dependence analysis as well as subsequent parallelization. The following checks currently exist in Cetus.
procedure \textit{DataDependenceAnalysis} ( \textit{PROG} )

\textbf{input} : Program representing all source files: \textit{PROG}
\textbf{output} : Data dependence Graph containing dependence arcs DDG

// Collect all FOR loops meeting eligibility
// Checks: Canonical, FunctionCall, ControlFlowModifier
\textit{ELIGIBLE\_LOOPS} = getOutermostEligibleLoops( \textit{PROG} )

\textbf{foreach} \textit{LOOP} \in \textit{ELIGIBLE\_LOOPS}

// Obtain lower bounds, upper bounds and loop steps
// for this loop and all enclosed loops i.e. the loop-nest
// Substitute symbolic information if available, Section 3.2.2
\textit{LOOP\_INFO} = collectLoopInformation( \textit{LOOP} and enclosed nest )

// Collect all array access expressions appearing within the
// body of this loop, this includes enclosed loops and non-perfectly
// nested statements
\textit{ACCESSES} = collectArrayAccesses( \textit{LOOP} and enclosed nest )

// Traverse all array accesses, test relevant pairs and
// create a set of dependence arcs for the loop-nest
\textit{LOOP\_DDG} = \textit{runDependenceTest}( \textit{LOOP\_INFO}, \textit{ACCESSES} )

// Add loop dependence graph to the program-wide DDG
// The program-wide DDG is initially empty
\textit{DDG} += \textit{LOOP\_DDG}

// return the program-wide data dependence graph once all loops are done
\textbf{return} \textit{DDG}

Fig. 3.2. High-level data dependence analysis algorithm. Collects eligible loop-nests and provides the necessary loop information for testing pairs of array accesses. A data dependence graph is provided as output.
Canonical Check

Data dependence analysis and loop parallelization in Cetus can only be applied in the case of FOR loops that have a canonical structure. This property is described in terms of the simplicity of the loop initialization statement, the loop condition and the loop increment. Not relying on a loop normalization pass, the dependence analyzer checks whether the initialization statement is a simple assignment statement assigning an integral or single variable value to the loop index variable. The loop condition must be a simple binary expression again with the loop index variable as one of the operands. The loop step must be a simple unary or binary expression and must increment or decrement the loop index variable by a constant integral value. In some cases, it might be possible to transform a WHILE loop into canonical FOR loop form, this is currently not handled in Cetus.

Function Call Check

Function calls are a significantly hard problem in optimizing compilers due to the inherent complexity of static-time interprocedural analyses. In Cetus, data dependence analysis considers only intraprocedural read/write dependence testing and can make no specific assumptions about information passed in and out of function calls. This requires that loops containing function calls be handled specially, by considering the information being passed in and out as well as the global variables being modified by the called procedure (callee).

Control Flow Modifier Check

A FOR loop with simple control flow is one in which there exists a single entry point into the loop following which the entire loop body is executed zero or more times. The index variable is incremented after every iteration of the loop and execution falls out of the loop once the loop condition evaluates to false. However, control flow
modifiers such as goto and return statements can cause execution to exit a loop early. This makes it impossible to parallelize the loop under consideration. For this reason, the Cetus infrastructure ignores loops that contain control flow issues for dependence testing as well. break and continue statements, although affecting control flow, allow for an outer-loop to be parallelized. These are hence, treated specially.

3.2.2 Symbolic Range Analysis Support

Cetus possesses advanced symbolic range analysis to identify values of symbolic variables at every program point [13]. These ranges are represented as integral lower and upper bounds, or alternatively, integral constants if the exact value of the variable can be determined. In the canonical checks mentioned above, the dependence analysis framework allows for symbolic values for all loop-related variables in question.

- **Lower loop bound**: If the bound is symbolic, obtain its value using range analysis. If it is an integral constant, use that value. If it is a range, use the lower bound of the range, thus conservatively accommodating for all possible loop iteration values.

- **Upper loop bound**: If the bound is symbolic, obtain its value using range analysis. If it is an integral constant, use that value. If it is a range, use the upper bound of the range, thus conservatively accommodating for all possible loop iteration values.

- **Loop increment**: If the step is symbolic, obtain its value using range analysis. If the step is an integral constant, use its value. If it is a range e.g. due to a symbolic value that is loop-variant, consider the loop non-canonical and hence ineligible.
void foo(double p[]);

int main(void)
{
    int i, j, k, ub;
    double t, a[10][10][10];
    for ( i=0; i<10; ++i )  ⇒  (ineligible: function call)
    {
        ub = 10;
        for ( j=0; j<ub; j++ )  ⇒  (ineligible: function call)
        {
            foo(a[i][j]);
        }
        for ( j=0; j<ub; j++ )  ⇒  (eligible)
        {
            for ( k=0; k<ub; k+=1 )  ⇒  (eligible)
            {
                t += a[i][j][k];
                a[i][j][k] = a[i-1][j][k];
            }
        }
    }
    return 0;
}

Fig. 3.3. Example code showing eligibility of loops. The loop containing a function call renders the outermost loop ineligible as well. The third and fourth loops at nesting levels 2 and 3 are eligible. Symbolic bounds such as ‘ub’ can be handled, provided the initialization, condition and increment expressions are canonical.

3.2.3 Information Collection

This wrapper forms the interface of the dependence analyzer with the Cetus HIR. The loop eligibility check filters out all loops that cannot be handled by the depen-
Fig. 3.4. The Data dependence wrapper traverses the IR and collects references to relevant IR information for local storage purposes. This information is then passed to the dependence test.

dence analyzer framework and eventually, by loop parallelization. For the eligible loops, the wrapper traverses the HIR and collects two pieces of information, loop information and array accesses as shown in Figure 3.4. Following the information collection phase, the dependence driver invokes dependence testing on pairs of array accesses to test for read/write and write/write dependences. To handle symbolic values inside of subscripts, we again use range analysis to obtain the relevant information at the given program point i.e. at the parent statement of each array access. The algorithm for this runDependenceTest driver is shown in Figure 3.5.

**Loop Information**

Data dependence analysis is a computationally intensive technique and can grow exponentially in time with increase in the number of loops, number of array accesses and the depths of loop nests. Collecting loop information includes traversing the Cetus HIR and storing local references to all necessary HIR objects and other information in custom data structures that are then used by the dependence tester.
This helps in avoiding HIR traversal each time loop-related information is required. Symbolic range analysis support described in Section 3.2.2 is used during this phase.

**Array Access Information**

All array accesses in a given loop nest must be tested against relevant array accesses irrespective of their nesting levels. The information collection wrapper stores all array accesses in list format for a given eligible loop nest. The local storage also contains relevant information regarding each array access such as whether it is a read or a write, its parent statement and its immediate enclosing loop.

**3.2.4 Dependence Test**

Following information collection and the HIR traversal process, all array accesses must be tested for dependences within the loop nest. The algorithm for this is outlined by `runDependenceTest` in Figure 3.5. Pairs of array accesses are tested for dependences. Having described the types of dependences earlier, we can infer that one reference from each pair of accesses being tested must be a `write` reference. Alias information must be used to determine if there are aliases referring to the same memory locations and hence must be paired with other aliased arrays for testing.

Over the last three decades, a number of mathematical formulations have been developed in order to solve systems of linear and non-linear equations for dependence analysis. Even in the case of linear or affine systems, proving dependence, i.e. finding an integer solution to the system, is a hard problem. An affine function is a linear function plus a constant [27]. The Banerjee inequalities provide a way of testing for independence for linear equations [27, 28]. Other subscript tests include the GCD test, SIV and MIV tests [26]. More complex tests that can handle non-linear subscripts include the Omega test [29] and the Range test [30]. Cetus currently supports implementations for the Banerjee and Range tests. The Banerjee test cannot han-
procedure runDependenceTest( LOOP_INFO, ACCESSES )

input : Loop information for the current loop nest LOOP_INFO
        List of array access expressions, ACCESSES

output : Loop data dependence graph LOOP_DDG

foreach ARRAY_1 ∈ ACCESSES of type write
    // Obtain alias information i.e. aliases to this array name
    // Alias information in Cetus is generated through points-to analysis results
    ALIAS_SET = getAliases( ARRAY_1 )
    // Collect all expressions/references to the same array from the entire list of accesses
    TEST_LIST = getOtherReferences( ALIAS_SET, ACCESSES )

foreach ARRAY_2 ∈ TEST_LIST
    // Simplify both expressions e.g. elimination of symbolic values
    // Obtain the common loops enclosing the pair
    // Especially relevant for non-perfectly nested loops
    COMMON_NEST = getCommonNest( ARRAY_1, ARRAY_2 )
    // Possibly empty set of direction vectors under which dependence exists is returned by the test
    DV_SET = testAccessPair( ARRAY_1, ARRAY_2, COMMON_NEST, LOOP_INFO )

foreach DV ∈ DV_SET
    // Create arc from source to sink
    DEP_ARC = buildDependenceArc( ARRAY_1, ARRAY_2, DV )
    // Build the loop dependence graph by accumulating all arcs
    LOOP_DDG += DEP_ARC

// All expressions have been tested, return the loop dependence graph
return LOOP_DDG

Fig. 3.5. Data dependence driver algorithm and wrapper for a single loop-nest. Traverses list of array accesses and builds the loop dependence graph.
dle non-linear subscripts, for example those created by symbolic values in the array subscripts that are coefficients of the loop index variable.

However, symbolic values can also appear in subscripts that are affine. In such cases, the dependence framework uses symbolic range analysis as described before in the form of constant propagation i.e. all symbolic variables with constant values are substituted into subscripts. This is highlighted as the expression simplification process in the `runDependenceTest` algorithm described in Figure 3.5.

After expression simplification, the pair of array accesses is passed to the testing framework. This phase of the algorithm is described by `testAccessPair` in Figure 3.6. Subscript partitioning is required to test corresponding array dimensions with respect to the correct loop index variables and under the corresponding loop constraints. Partitioning creates pairs of array dimensions from the subscripts. These pairs can be tested together as in the case of the Omega test, but must be tested individually for the Banerjee and Range tests. The details for subscript partitioning are provided in [26]. The common enclosing loop nest provides the direction-vector constraints for the pair being tested. Different combinations of direction-vectors are traversed as outlined in the algorithm in [28]. Each partition pair is individually tested by the Banerjee or Range test. The Banerjee inequalities show independence by proving that a solution does not exist under the constraints derived from loop information and direction-vector information. Symbolic values that appear during the test computations are handled through symbolic analysis and integral values are derived if possible. The output of the test is a dependence or absence of dependence from the `source` array access to the `sink` array access for the given direction vector. The Range test handles non-linear subscripts in addition to the cases mentioned above through advanced symbolic analysis. It does not solve for systems of equations, but tests for overlapping access regions instead. The set of direction vectors returned from the tests is then used to create a `dependence arc` from the source to the sink array access and subsequently, a dependence graph is composed of these arcs.
procedure testAccessPair( A1, A2, COMMON_NEST, LOOP_INFO)

    input : Pair of array accesses to be tested A1 and A2
            Nest of common enclosing loops COMMON_NEST
            Information for these loops LOOP_INFO

    output : Possibly empty set of direction vectors under
             which dependence exists DV_SET

    // Partition the subscripts of the array accesses into dimension pairs
    // Coupled subscripts may be handled, see Section 3.2.4
    PARTITIONS = partitionSubscripts( A1, A2, COMMON_NEST )

    foreach PARTITION ∈ PARTITIONS
        // Depending on the number of loop index variables in the partition,
        // use the corresponding test. Details in [26]
        // zero index variables ZIV
        if( ZIV )
            DVs = simpleZIVTest( PARTITION )
        // single or multi-loop index variables: SIV, MIV
        else
            // traverse and prune over tree of direction vectors, collect DVs where
            // dependence exists
            // The traversal is described in [28] and not shown here
            foreach DV ∈ DV_TREE using prune
                // In Cetus, the MIV test is performed using Banerjee or Range test
                DVs += MIVTest( PARTITION, DV, COMMON_NEST, LOOP_INFO )
            // Merge DVs for all partitions
            DV_SET = merge( DVs )

    return DV_SET

Fig. 3.6. Dependence test wrapper that handles actual partitioning
and traversal of the pairs of subscripts that are being tested. Handles
low level direction-vector traversal and calls the actual dependence
test e.g. Banerjee test
3.2.5 Data Dependence Graph (DDG)

A data dependence graph stores dependency information in the form of dependence-arcs from source array accesses to sink array accesses, thus describing the structure of dependencies. The arcs contain direction-vector and dependence type information, which characterize each dependence. The array access data structure described in Figure 3.4 is used as information storage for nodes of this graph. A loop-nest based graph is created from the information returned by the tests, which is then used to build a program-wide data dependence graph. The runDependenceTest algorithm in Figure 3.5 shows the construction of this graph from data returned by the tests. The data dependence graph provides an API to help other analysis and transformation passes to query dependence related information. Sub-graphs for specific loop nests can be extracted, as described in the Loop Parallelization pass explained later in this section. API functions are also provided to query statement-based dependence information, which is generated by internally accessing information stored within the nodes. Such information can be particularly useful for transformations such as loop-tiling that require statement reordering.

3.3 Alias Analysis

Alias analysis is a compiler technique that identifies two or more variables referring to the same memory location. As opposed to dependence analysis, which identifies dependencies of memory locations that flow from one program point to another, alias analysis identifies aliased variables at a specific program point. It is important to note that for the above described dependence analyzer, it is essential to identify aliased variables in order to determine the set of array accesses that must be tested against each other for dependences within the loop. Various alias analysis techniques have been proposed for optimizing compilers over the last few decades. An important classification of these analyses is flow-sensitivity and context-sensitivity.
- **Flow-Sensitivity**: Static time analyses traverse the source IR and collect necessary information that feeds as input to the specific analysis. The output of the analysis can be a conservative view of the entire program, or the analysis can use the input to provide an output for every execution point in the program. Such an analysis that considers the order of execution and provides output for every execution point in the program is a flow-sensitive analysis. Flow constructs such as branches and loops can be incorporated into flow-sensitive analyses to provide more precise results.

- **Context-Sensitivity**: The concept of context-sensitivity relates to interprocedural static-time analyses. Interprocedural analysis takes into account the effect of function calls by analyzing the body of the called procedure (callee). However, a single procedure can have more than one calling sites in the program, each of which can be considered as a context of the call. A context-sensitive analysis is one which takes into account the context of the call while providing interprocedural updates. Context sensitivity can have varying degrees of complexity in terms of the depth of the analyzed function calls and the storage of analysis updates corresponding to different contexts.

Being flow and context-sensitive makes an analysis much more complex and computationally intensive, but also more precise. In the C programming language, aliases can mainly be created through the use of pointers and through explicit pointer assignment using ‘&’ and ‘*’ operators, and also, as is commonly used, through passing arguments by reference into function calls.

### 3.3.1 Conservative alias analysis with manual input

The first alias analysis implementation in Cetus was flow and context-insensitive, thus collecting a conservative set of aliased variables for the entire program. An extension was added to this analysis to allow the user to provide more precise information through manual analysis. The new analysis provides, via command-line options, the
functionality to specify whether the program follows the FORTRAN alias assumption i.e. the assumption that arguments to a function call are not aliased to each other and to global variables. The default is considered to be the C assumption where all parameters for a function are aliased to each other and to all global variables accessed within that procedure. The ability to provide more precise information that can be interpreted as being interprocedural input, helped improve parallelization results by manually analyzing the benchmarks and by identifying that all arguments to function calls incurred disjoint accesses inside the callee.

3.4 Points-to analysis

The manual/auto hybrid analysis described above requires programmer input to improve the results of parallelization. Automatic static-time analysis can be used to substitute programmer input and achieve similar parallelization results. Points-to analysis is a technique that helps identify pointer relationships during program execution and these relationships can be critical in identifying alias sets for variables in the program. This section describes the implementation of an intra-procedural points-to analysis in Cetus and an overview of its extension to inter-procedural analysis in order to automatically identify alias sets.

A pointer type variable in C (pointer in short) exists specifically to hold the memory address of another memory location. A pointer is said to point to the location or reference the location whose address value it contains and the act of accessing the value at the pointed-to location is known as dereferencing the pointer. Pointers are powerful programming constructs allowing low-level manipulation of data values during program execution through techniques such as pointer arithmetic and allocation of memory during program execution i.e. dynamic memory allocation. The goal of the points-to analyzer is to identify, at every program point, the set of named or unnamed memory locations that a pointer variable may point to. Points-to relation-
procedure PointsToAnalysis( PROC )

input : Interprocedural Node PROC (a Procedure)

output : Map(Statement, PointsToDomain) DOMAIN_MAP

// Interprocedural Node: The node representing a Procedure in the program call graph
// Statement: Statement construct representing program execution unit
// PointsToDomain: Storage for points-to information at every program point

// Build the control flow graph for this procedure
// Sort the nodes in reverse-post order for iterative data flow analysis
CFG = buildControlFlowGraph( PROC )
WORK_LIST = topologicalReversePostOrder( CFG )

// Iterate over work-list and return map for points-to information
DOMAIN_MAP = iterateWorkList( WORK_LIST, PROC )

return DOMAIN_MAP

Fig. 3.7. High-level intraprocedural points-to algorithm. Uses flow-sensitive approach by building a control flow graph and returns a map from statement to points-to domain for the entire procedure.

ships obtained as a result, can be used to create alias sets for that program point by identifying different pointers, or aliases, pointing to the same memory locations.

The Cetus points-to analysis is based on the design described in [31]. A flow-sensitive approach is used for analyzing intra-procedural pointer information. Within a procedure, pointer relationships are created through explicit pointer assignment. Figure 3.7 gives the top-level pseudo-algorithm for the implementation of the points-to analyzer in Cetus. The following sections describe i) iterative data flow updates using control flow information, ii) handling explicit pointer assignment for creating the data-flow information and iii) data structures used to store relationship information.
procedure iterateWorkList( WORK_LIST, PROC )

input : Reverse-post ordered work list containing
statement-based CFG nodes WORK_LIST
Interprocedural Node PROC (a Procedure)

output : Map(Statement, PointsToDomain) DOMAIN_MAP

while ( WORK_LIST ≠ φ )

// IN, OUT are of type PointsToDomain and constitute of data-flow information
// Extract the first node from the list and analyze it for points-to updates
CURR = removeFirstNode( WORK_LIST )
// merge() is used to combine points-to data from all predecessors
foreach pred ∈ CURR.preds

IN = merge( pred )
// If the node represents a Statement, populate result map
if ( CURR is a Statement )

DOMAIN_MAP.put( CURR, IN )
// Perform data-flow update
OUT = pointsToUpdate( PROC, CURR, IN )
// Reinsert successors into work-list only if there is an update
LAST = CURR.getDomain()
if ( OUT ≠ LAST )

foreach succ ∈ CURR.succs

WORK_LIST.insert( succ )
// Save output for the current node for the next time it is needed
CURR.putDomain( OUT )
// if work-list is empty, no more updates, return the map

return DOMAIN_MAP

Fig. 3.8. Iterative traversal of the reverse-post ordered work-list. This traversal is used to analyze the control flow graph for points-to updates. The merge operation is defined in [31].
procedure pointsToUpdate( PROC, CURR, IN )

input : Interprocedural node PROC (a Procedure)
        Control flow graph node, CURR (a Statement)
        Input PointsToDomain information IN

output : Output PointsToDomain information OUT

// Traverse the statement in the Node for handling points-to
// information updates
// Perform normalization on the statement to allow generic handling
// i.e. in the form of assignments and function calls as shown below
STMT = (Statement)normalize( CURR )

// Handle cases such as:
// x = &y
if ( STMT contains AssignmentExpression &&
     LHS is pointer_type )
    OUT = processAssignment( PROC, STMT, IN )

// Handle cases such as:
// foo()
else if ( STMT contains FunctionCall )
    OUT = getInterProceduralInfo( PROC, IN )

return OUT

Fig. 3.9. Handling of assignment statements and function calls during
data-flow analysis. Through normalization, most statements can be
converted to conform to one of these two constructs. See Section 3.4.7
for normalization details.
3.4.1 Iterative work-list based control flow analysis

Cetus implements a flow-sensitive points-to analysis. Ignoring the inter-procedural aspect of the analysis, a control flow graph (CFG) is obtained for each procedure undergoing intra-procedural analysis. A CFG is a graph representation of all the paths that might be traversed during program execution. In Cetus, the CFG is statement based i.e. each control flow node corresponds to a single execution unit. Slight departures from this model are necessary for modeling the control flow around loops and branch statements. Cetus’ points-to analyzer handles flow-sensitivity differently from the analysis described in [31]. With the statement-based CFG, compound control-flow blocks such as loops and branch statements do not need to be handled separately but are an integral part of the CFG traversal and points-to update process.

An efficient method for implementing forward data-flow analysis on a CFG is reverse-post ordering of the nodes, which ensures that all parents of a node are analyzed before the node itself [32]. This ordering also favors the worklist based approach, where all nodes in the list are analyzed at least once. Successors of each node are re-inserted into the work-list if and only if there is a change in the points-to relation for the current node from the previous relation that was assigned to the node. The reverse-post ordering and work-list based approaches seamlessly incorporate the fixed-point iteration necessary for handling points-to relationship updates caused by loops. Loops are modeled at static-time through iterative analysis. A fixed-point is reached when the information generated by the loop converges to a single point and undergoes no further changes on additional iterations of the loop. This fixed-point can be used as the loop’s exit representation.

Points-to information is determined at each program point in the form of Domain information. Each statement is mapped to the points-to information that exists before that statement might be executed. The map structure provides a quick API for obtaining points-to information at every program point. A high-level overview of the work-list traversal is provided in Figure 3.8 and Figure 3.9 shows the two main
constructs handled during the data-flow analysis, which are described in the following sections.

### 3.4.2 Points-to Domain and Points-to Rel

```c
int main(void) {
    int *a, *b, c;

    /* [] */
    a = &c;
    /* [(a,c,D)] */ \(\Rightarrow\) (PointsToDomain containing a single PointsToRel)
    c = 10;
    /* [(a,c,D)] */
    b = a;
    /* [(a,c,D), (b,c,D)] */ \(\Rightarrow\) (PointsToDomain containing multiple PointsToRels)
    return 0;
}
```

Fig. 3.10. Simple example showing PointsToRel and PointsToDomain, printed by Cetus before their corresponding program points.

As per the design described in [31,33], a points-to relationship can be represented using three pieces of information, i) the head of the relationship which is a pointer, ii) the tail of the relationship which is a location referenced by the head and lastly, iii) whether the relationship is definitely or only possibly true at that program point. These pieces of information are represented in a data structure called PointsToRel in Cetus. The head and tail of the relationship are represented using Symbols, which as described in Section 3.1.1 provide the symbol table interface for statically allocated, named memory locations. For unnamed locations such as dynamically allocated memory, string literals and null pointer values, we use specially created Symbol
information. More details are provided in Section 3.4.3. The definite or possible status helps identify the validity of the relationship, which is simply a boolean value. More details regarding the use this status information are provided in [31]. Cetus provides an interface called Domain that can be used to represent analysis information. We use a derived class, PointsToDomain, that holds a set of PointsToRel and that is mapped to every Statement in the program. This PointsToDomain provides the implementation specific to the merge, union, diff and kill operations necessary for the data-flow updates. These appear in Figures 3.8 and 3.11. Thus, we can concretely represent points-to information at every program point in a flow-sensitive environment. Figure 3.10 serves, at this time, only to provide a view of the relationship and its association with statements.

### 3.4.3 Pointer Symbols: Stack allocated and heap allocated variables

In C programs, program variables and aggregate structures can either be allocated a chunk of memory at compile time or during program execution. Variables declarations provide named locations for statically allocated memory and these variables are allocated on the program stack. As opposed to the above, C allows declarations of pointers at compile time which can be assigned with addresses of memory locations that are allocated at runtime e.g. through standard library calls such as malloc. Such memory is allocated on the program heap, is accessible for the entire duration of the program unless it is freed and is unnamed. A pointer must be used to dereference dynamically allocated memory. C allows for the specification of the type and size of dynamically allocated memory. In the case of aggregate data structures, each named location can be used to refer to more than one named or unnamed memory location.

In order to represent these various types of locations, points-to analysis makes use of Symbols. In Cetus, symbols represent the symbol table link for all declared variables and can hence be guaranteed to represent a unique variable within a given program scope. Arrays in C are aggregate variables with a single name and multiple locations
are accessed through the array subscript program construct. Hence, the concept of named locations is complex in the case of arrays. As a result, handling explicit pointers to different elements of the array significantly increases the complexity of points-to analysis. Consequently, in Cetus’ points-to analysis, arrays are treated as single locations, similar to scalar variables, and all relationships corresponding to pointers to arrays or arrays of pointers are deemed as possible relationships. Aggregate structures such as ‘structs’ are precisely handled as a combination of scalar and array variables. The Symbol interface described above cannot exist for dynamically allocated memory as the locations are unnamed. In order to handle such heap-allocated locations, we provide a common interface called AbstractLocation that provides a naming scheme based on the allocation site of the dynamically allocated memory. This is a departure from the design described in [31], where the entire program heap is considered as a single location. In the Cetus analyzer, it is possible to distinguish between memory locations allocated from different program statements. This helps in improving alias analysis, as we shall see later. AbstractLocation can be used to determine the type of dynamically allocated memory when allocated via functions such as malloc and in other situations such as representing the unnamed location used to store a String Literal pointed to by a char pointer.

3.4.4 Pointer assignment

Pointer information is either initialized during the declaration section inside a well-defined scope or generated through assignment statements during execution. The pointer assignment transfer function forms the core of the intra-procedural points-to analyzer. The first step in the update involves identification of the locations participating in the assignment. Following this, data-flow updates such as gen and kill must be applied to obtain the output points-to domain resulting from the assignment statement. Figure 3.11 provides an outline of the algorithm used for processing pointer assignments.
procedure processAssignment( PROC, STMT, IN )

input : Interprocedural node PROC
         Statement contributing to points-to update STMT
         Input PointsToDomain information IN

output : Output PointsToDomain information OUT

// Compute set of L-locations i.e. locations that consume values
LLOC = computeLLocations( STMT, IN )

// Compute set of L-locations to kill, can be different from
// L-locations because of definite(D) and possible(P)
// relations
LLOC_KILL = computeLLocationsKill( STMT, IN )

// Compute set of R-locations i.e. locations that supply values
// If rhs is function call, use interprocedural information
if ( RHS contains FunctionCall )
    IN = getInterproceduralInfo( PROC, IN )
    RLOC = getFunctionReturnVal( FUNC_CALL )
else
    RLOC = computeRLocations( STMT, IN )

// If any of the above computations for LLOC, LLOC_KILL
// and RLOC fail due to any reason
// return the universe domain, see Section 3.4.5
if ( computation failed )
    return UNIVERSE_DOMAIN

// Handle data-flow updates
GEN = createGen( LLOC, RLOC )
KILL = createKill( LLOC_KILL )
OUT = GEN \cup ( IN \setminus KILL )

return OUT

Fig. 3.11. processAssignment handles the crucial implementation for the data-flow transfer function. LHS and RHS locations are computed, and gen, kill, union(∪) and diff(−) operations are performed during the assignment update.
Computing L-locations and R-locations

Only pointer assignment statements must be considered for updates by the points-to analyzer. The expressions on the left-hand side (LHS) and right-hand side (RHS) of the assignment may be complex and can involve multiple dereferences, struct member accesses and so on. The L-locations and R-locations correspond to the actual locations referenced by variable references resulting from the expressions in the LHS and RHS. We use the same design base for these locations as in [33]. Each L-location and R-location has a corresponding boolean value associated with it to determine the type of the relationship that the location generates. Thus, from our previous discussion regarding arrays and their treatment as scalar variables, it should be evident that an array access must correspond to a L or R location value represented by the Symbol of the array, and must be assigned a boolean value of FALSE i.e. only a possible relationship. These boolean values help determine the relationship of the final locations when accessed via long pointer chains i.e. expressions that contain multiple dereference operations or member access operations. The location analyzers can return more than one L-or-R-location as various pointers along different chains might possibly point to more than one specific location. In the event of insufficient information to compute the L-or-R-locations using pointer dereferencing and input data, the analysis returns the UniverseDomain, which is described in Section 3.4.5.

Dataflow equations

Forward data flow analysis uses a standard transfer function defined specifically for the analysis to propagate information following an update for the current node. In data-flow terminology, the gen information is represented by new PointsToRels created by the pointer assignment. kill is the set of all symbols for which new relationships are being created and whose earlier PointsToRels must all be destroyed, unless the new relationships are only ‘possible’ relationships. At each program point,
int **d, *a, *b, c;

/* [] */
d = &a;
/* [(d,a,D)] */
// b = a
// But a is uninitialized
// This results in Universe
b = (*d);
/* [UNIVERSE] */
a = &c;
/* [UNIVERSE] */
c = 10;
/* [UNIVERSE] */
if ( ( **d)-(b) == 0 )
{
    ;
}

Fig. 3.12. Example code showing conservative points-to update via the UniverseDomain. Such cases mostly arise due to conservative assumptions in other situations such as non-normalized or unknown expressions or unhandled function calls.

the input information is that which is obtained via a merge of all the predecessors’ points-to relationships. These equation formulations are described in detail in [31].

3.4.5 Null Domain and Universe Domain

Forward data-flow analysis entails monotonicity of the transfer function that provides an update to the analyzed information [34]. Monotonicity requires that the data information grows following an update, this is essential so that the data-flow analysis
converges in the case of fixed point iteration. In order to recognize this requirement, the points-to analyzer in Cetus uses two additional types of domains: NullDomain and UniverseDomain. The NullDomain represents an initial or startup value of the information at every program point before the analyzer has begun its forward traversal process. On the other extreme, there must be a set which represents the entire universe of variables and pointers, which could also be viewed as the most conservative location that any pointer can point to. The UniverseDomain represents this information. In the event that the analyzer is unable to accurately apply its transfer function, the analysis must default to a conservative assumption rather than moving on with incomplete or incorrect information. An example code section that might result in the UniverseDomain is shown in Figure 3.12. It is important to note that this is the most conservative assumption and no updates are possible to the universe, irrespective of the program point or the type of update. The hierarchy of Domain representations for points-to information is shown in Figure 3.13.
3.4.6 Function Call handling: Interprocedural Overview

Cetus also implements an interprocedural framework for points-to analysis that uses the intraprocedural analysis described above. Handling a function call inside of intraprocedural analysis is essential for forward propagation of information. In the absence of interprocedural analysis, intraprocedural assumptions cannot be made regarding points-to information generated or killed by a callee. Conservative function call handling may help avoiding the *UniverseDomain*. With the help of interprocedural information, however, points-to analysis can obtain mapped and un-mapped information at procedure boundaries, and obtain accurate results through intraprocedural analysis of the callee. Functions can return pointer values as well, a common example being `malloc`. The current points-to analysis handles this using the assignment paradigm, where the RHS is a function call and the LHS is the pointer that the return value is being assigned to. This is represented in the `processAssignment` algorithm in Figure 3.11.

3.4.7 Normalization

In the Cetus points-to analyzer, we do not enforce the normalization described in [31]. We handle multiple dereferencing and pointer chains through normal handling as far as the points-to relationships can be determined. If the depth of the pointer-chain is too large, or if information is not available at any single level, the analyzer defaults to the *UniverseDomain*. However, we do enforce normalization of how the dereference chains are represented. Some examples of normalization are presented in Figure 3.14.

3.4.8 Applications

The use of pointers is widespread in C programs. As mentioned before, the use of ‘&’ and ‘*’ operators as well as pointer parameter-passing create huge complexity in
// array access
int a[10][10][10];
foo(a[0][10]); ⇒ foo( &(a[0][10][0]) );
... = a; ⇒ ... = &(a[0][0][0]);

// field access
struct {
  int x;
  int b[10][10];
} *y;
... = *(y->x); ⇒ ... = *( (*y).x );
... = y->b[0]; ⇒ ... = & ( (*y).b[0][0] );

// pointer arithmetic
int c[10];
int *p = c; ⇒ int *p = &c[0];
p += 2; ⇒ p = &p[2];
p = c+5; ⇒ p = &c[5];

Fig. 3.14. Expression normalization examples for points-to analysis. These internal and temporary transformations help simplify the analyzer and help handle certain corner cases.
the way different pointers can be used to refer to different locations. Adding to this complexity is the issue of pointer arithmetic, which is of notable importance when using pointers to access aggregate structures such as arrays. Providing points-to information can hence be critical to many other compile-time analyses. The compiler can be prevented from making worst-case assumptions using more accurate information and thus provide fruitful results. Some examples of these benefiting analyses are described below.

**Alias and Data-dependence Analysis**

Alias information has traditionally been represented by sets of alias-pairs. Points-to analysis can be used to generate accurate alias sets at every program point. Alias sets can be generated with relative simplicity for different cases of points-to information [31]. For procedures that accept parameters that are arrays, these parameters are truly pointers to the array being passed in. Points-to analysis can provide accurate information regarding these arrays, thus facilitating better alias analysis within the called procedure (callee). From the discussion on dependence analysis, the importance of accurate alias information is evident. References to memory locations through aliases must be tested against each for loop-carried dependences, thus affecting the results of loop-parallelization.

Points-to information can help identify aliased arrays for an entire loop. Through interprocedural analysis, it is possible to determine the outer-scope arrays that array-parameters of a function may be pointing to. Aliasing between different parameters of the function and between parameters and global variables is detected by the points-to analyzer, thus providing the dependence analyzer with even more accuracy. This results in fewer spurious dependences in the program DDG and the loop parallelization pass described in Section 3.5 can parallelize more loops. Figure 3.15 shows the improvement in number of parallelized loops obtained across a range of NPB and SPEC OMP2001 benchmarks using different types of alias analysis in Cetus. The *all-alias*
Fig. 3.15. This figure shows the difference in the number of loops parallelized by Cetus under different alias assumptions or analyses. *all-alias* is where no assumptions are made about function arguments and conservative alias information is used. *argument-noalias-global* allows manual input of information showing that arguments are not aliased to each other or to global variables. Accurate interprocedural points-to analysis, *alias-advanced-interproc* gives close to the best possible alias results and hence, close to the FORTRAN assumption results on 9 out of 10 benchmarks.

results show the number of loops auto-parallelized by Cetus using the conservative global alias analysis without any flow-sensitivity or interprocedural information. The *argument-noalias-global* seeks to imitate the FORTRAN assumption and is described in Section 3.3. This result was achieved through manual program analysis of the benchmarks and the fact that none of the arguments to function calls had overlapping accesses inside the callee that would prevent parallelization. The *alias-advanced-interproc* Cetus option performs precise interprocedural points-to analysis and uses this result for alias information. Accurate analysis helps achieve the best results in *9 out of 10 benchmarks* i.e. they all match the results achieved using the FORTRAN assumption manual input. *BT* is a special case that requires an additional simple
analysis to disambiguate sub-arrays. In $MG$, accurate points-to analysis falls short of achieving the goal. This is because the benchmark also has significant occurrences of sub-array passing to the same subroutine. However, unlike $BT$, the simple disambiguation analysis described above is unable to provide results in this case because all the arrays in $MG$ are allocated dynamically and are passed into the subroutines as explicit pointers. The C standard lacks sufficient information that would allow assumptions to be made regarding the legal range of memory accessed via these pointers inside the callee. This lack of information to make assumptions about the legality of the input program prevents further parallelization.

**Array Section Analysis**

Array sections describe the access patterns corresponding to array elements for a specific program section. Compiler passes such as array privatization $BBD09$ that deal with array access patterns could benefit greatly from section information. Pointer information can be crucial for array section analysis in a language like C where pointers can be used to access arrays as if they were declared as arrays themselves. Interprocedural points-to information related to arrays can subsequently help interprocedural array section analysis, thus providing even more accurate information to other compiler analyses.

**Constant Propagation**

This is a form of data-flow analysis where variables having constant values at specific points in the program can be replaced by their constant values so as to allow other compiler analyses to move ahead, without the need for symbolic manipulation. The example in Figure 3.16 shows that without pointer information, the compiler would find it impossible to evaluate the value of the if condition. This would mean that the compiler will never know that the condition always evaluates to true in this case. This could seriously affect other analyses and transformations as well.
Fig. 3.16. Example code showing importance of points to analysis for techniques such as constant propagation. The if condition in this example would evaluate to false, but this would be impossible to determine without points-to information. Optimizations such as constant propagation and hence, dead-code elimination can thus benefit.
3.5 Loop Parallelization

Cetus is a source-to-source translator. As such, automatic parallelization strategies are focused on coarse-grained parallelism i.e. parallel loops. Introduced earlier in Section 1, the OpenMP parallel programming model supports insertion of source-level directives. The backend compiler then uses these directives to generate parallel code. Cetus takes advantage of the OpenMP parallel programming model to parallelize loops using the `omp parallel for` directive [8]. This directive supports the DOALL model of parallelization, which entails that all parallel iterations of the loop must execute simultaneously. This requires the absence of loop-carried dependences for parallel iterations. Loop-parallelization is achieved using the results produced by the dependence analyzer. The high-level algorithm for LoopParallelization in Figure 3.17 describes the traversal of eligible loops, extraction of the data dependence information and query of this information for automatic parallelization.

3.5.1 Interface to DDG

The program-wide DDG is the output of the dependence analyzer and incorporates a set of all dependence-arcs that contain information related to the source and sink array accesses, their relevant enclosing loops as well as parent statements. Most importantly, the arcs contain a direction vector corresponding to the dependence and the type of that dependence. Loop parallelization must analyze an entire eligible loop nest at the same time in order to determine various dependence loop levels as well as handle dependences across different nesting levels in the case of non-perfectly nested loops. The DDG provides an interface to extract, from the program dependence graph, a set of arcs that are relevant only to the loop nest under consideration.
procedure LoopParallelization( PROG )

input : Program representing all source files: PROG

side-effect : Annotate loops with parallel directive

// DDG information for auto-parallelization is available for eligible loops, see Figure 3.2
ELIGIBLE_LOOPS = getOutermostEligibleLoops( PROG )

foreach LOOP ∈ ELIGIBLE_LOOPS

    LOOP_DDG = extractLoopSubGraph( PROG.DDG )

    // Traverse this loop-nest from the outermost to innermost loop

    foreach L ∈ LOOP+INNER_LOOPS in order

        parallel = TRUE

        // Check for scalar dependences using Privatization and Reduction analysis

        if ( scalar-var-defined ∉ Private( L ) ||
            scalar-var-defined ∉ Reduction( L ) )
            parallel = FALSE

        // Else check for array dependences. Direction vector DV is of the form:
        // <,=,> where each direction corresponds to a loop in the nest in that order

        else

            foreach DV ∈ LOOP_DDG

                // Check for non-equal loop dependence direction

                if ( DV.DIRECTION( L ) ≠ EQUAL_DIRECTION )
                    if ( DV.source && DV.sink
                        ∉ (Private(L) || Reduction(L)) )
                        parallel = FALSE

                // Non-equal dependence causes serialization. Serializing the outer
                // loop covers dependences for inner loops. Algorithm detail in [26]

                LOOP_DDG.delete( DV )

            if ( parallel )

                cetusParallelAnnotation( L )

Fig. 3.17. High-level loop parallelization algorithm. Uses the data dependence graph and privatization and reduction information. Direction vector traversal and elimination of covered dependences detailed in [26].
3.5.2 Handling scalar dependences

The dependence analyzer produces the output DDG containing dependence information for array accesses only. Cetus employs a different strategy for dealing with scalar dependences in the case of parallelization. Having employed the techniques of privatization, reduction and induction variable substitution [14], it is implicit that all scalars that are neither private nor reduction variables for the given loop must necessarily incur a dependence across different iterations of that loop. Thus, instead of a separate scalar dependence/flow analysis, we use the results of privatization and reduction to determine the effect of scalar dependences while parallelizing a loop.

3.5.3 Direction Vectors and Serializing Outer Loops

Having extracted all dependence arcs for a loop nest from the program DDG and having made sure that no scalar dependences are present for all the loops under consideration, the loop parallelizer considers each direction vector individually. The loop nest is traversed from outermost loop inwards. For each nesting level, every direction vector is queried to obtain the direction for the dependence for that loop. If a non-equal direction i.e. loop-carried dependence exists, the loop must be serialized. The array references that undergo the loop-carried dependence are checked for privatization or reduction with respect to that loop. If both source and sink are part of either set, the loop can be made parallelizable. Array privatization is supported by the OpenMP standard, but array reduction is not and hence, the analyzer is equipped to ignore array references that might appear as reduction variables. Having serialized an outer loop, all its non-equal dependences may be deleted thus making it possible that inner loops with non-equal dependences may still be parallelizable. The algorithm for testing dependences based on direction vectors is described in [26] and is outlined in Figure 3.17.
3.5.4 Cetus Annotations

The output of loop parallelization is provided in the form of internal Cetus Annotations. These annotations resemble the *omp parallel for* loop directive, except that privatization and reduction variable annotations are stored separately. Annotations in Cetus can be attached to specific statements that they apply to, as is done in the *for* loop case. An output OpenMP pass is then used to translate the internal Cetus annotations into *omp* directives and provide the necessary output.
4. AUTOTUNING OF PARALLEL PROGRAMS

The Cetus translator supports identification of loop level parallelism and OpenMP-directive based source code generation. The analyses and transformations described in Section 3 are compile-time strategies that purely check for the legality of parallelization i.e. parallelizability. As described earlier, OpenMP uses low-level libraries such as Pthreads to generate parallel executable code. Thus, an `omp parallel for` directive results in the loop iteration space being partitioned across parallel threads in order to execute the iterations in parallel. The backend compiler inserts code to fork threads at the beginning of a parallel section, which could be a simple code section or something specific such as a loop. Having executed the code in parallel, the threads must then join, possibly committing data back to the shared address space for subsequent serial execution. This is the fork-join model of parallelism and is implemented by OpenMP. The model can be improved in efficiency by reducing the number of forks and joins and using thread wake and sleep options instead.

Thread fork-join implementations create parallelization overheads, where the underlying system must spend extra cycles in order to branch out new threads and perform associated operations such as allocating thread stack space. Loop parallelization can lead to significant overheads; in the common case, even an outermost loop is executed many times during program execution, thus accruing fork-join cycle overheads. Parallelizing an inner or nested loop leads to even more performance degradation, as overheads are incurred on each new execution of the inner loop. In some cases, the overheads due to parallelization can result in the parallel program performance being much worse than serial execution.

Cetus has no compile-time heuristics or model driven approach to predict the behavior of parallelized loops and hence, it is expected that overheads associated with parallelizing small, inner loops would be significant. Automatic parallelization
Table 4.1
Number of loops automatically parallelized by Cetus (only the loop that is parallel and not enclosed by parallel loops is considered) and those already present in the input are shown in this table for a subset of the NAS Parallel and SPEC OMP2001 benchmarks. The number of Cetus-parallelized loops forms the search space for our tuning approach. Parallel loops that are nested inside outer loops can produce significant overheads due to repeatedly spawning and joining parallel threads. Selecting the right loops out of all the parallelized loops is crucial to obtaining better performance.

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
<th>equake</th>
<th>art</th>
</tr>
</thead>
<tbody>
<tr>
<td>CetusParallel</td>
<td>72</td>
<td>16</td>
<td>6</td>
<td>11</td>
<td>43</td>
<td>12</td>
<td>102</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>HandParallel</td>
<td>54</td>
<td>24</td>
<td>1</td>
<td>6</td>
<td>29</td>
<td>12</td>
<td>70</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

must be supported by a decision making framework in order to determine a subset of parallelizable loops that is actually beneficial to program performance.

Table 4.1 shows the number of loops parallelized by Cetus at a single level in the loop nest i.e. the outermost possible level. Some benchmarks with large, but regular loops such as BT, LU and SP from the NAS benchmark suite exhibit a larger percentage of computationally-intensive parallel loops, whereas the number drops for others i.e. for programs that exhibit loops with function calls, irregular subscripts or control flow issues as they are not amenable to auto-parallelization.

The tuning framework described in this thesis makes the following contributions:

- A compiler and runtime system that detects parallel loops in serial and parallel programs and selects the combination of parallel loops that shows the highest performance.

- Finding profitable parallelism can be done with a generic tuning method. The Combined Elimination Algorithm, which was previously used to tune a generic set of compilation flags [35] is used in this framework.
Fig. 4.1. High-level overview of the offline empirical tuning system, Cetune. A search algorithm navigates through the space of possible optimization variants. It makes a decision on the next suitable optimization variant that the system should try. This variant is handed to the version generator in order to be compiled. Performance of each version is measured using execution time, the result of which feeds back into the search-space navigation algorithm, which determines the next suitable variant, until a convergence criterion is satisfied.

- A method that can be applied on a section by section basis, to an entire program. This is used to provide fine-grained tuning of parallelizable loops in the program.

- On a set of NAS and SPEC OMP benchmarks, this tuning method can deliver program performance that improves through automatic parallelization (of both serial and parallel program) and does not degrade below that of the original program in almost all cases.

4.1 Cetune: Automatic Tuning Framework

This section describes the automatic tuning system. The core part of the system is comprised of i) navigation through the search space, ii) version generation and iii) empirical evaluation of performance, as shown in Figure 4.1. Each of these is described in detail in the following sections.
4.1.1 Methodology

The automatic tuner supports the automatic parallelizer in providing a parallelized version of the code that does not decrease performance below that of the original code. This can be achieved by identifying loops that, when executed in parallel, contribute significantly to fork-join overheads or suffer from poor performance due to issues such as memory locality or load imbalance. In most cases, the size of the parallelizable loop body is too small or its iteration space is not large enough to amortize the cost of (micro)thread creation and termination. By serializing these loops during execution, the above overheads can be minimized, thus leading to more favorable execution time.

Static time models [36, 37] provide the compiler with a performance estimate based on which certain optimizations would lead to better run time. These models can be overly simplified or in many cases too complex, especially while modeling for different optimizations. They also fail to accurately capture interactions between various optimizations. Instead, the Cetune framework measures whole program execution time, which is a simple, but comprehensive metric for considering all effects of various optimizations. The work described here, however, only considers parallelization/serialization of loops in the optimization search space. No other optimizations are considered.

In order to consider the overall effects of different combinations of parallel or serial loops on the program execution time, an offline empirical tuning method is used. Empirical tuning, as described in Section 2, involves the actual execution of the application and run-time measurement of metrics as opposed to a model-based approach which runs at compile-time. Offline tuning performs all operations in outside-of-production runs and provides a final optimized solution that can be used for the production runs. This is the opposite of modifying optimization strategies during actual production runs, based on run-time performance history, which is called dynamically adaptive optimization. While the research makes no claims about the
superiority of offline tuning versus dynamically adaptive optimization [38], offline tuning performs well for most of the present set of benchmarks.

4.1.2 Tuning Algorithm

Empirical tuning can make use of the most powerful performance metric, which is execution time. But an exhaustive search of all the possible combinations of optimization variants can be prohibitively expensive in this case. The search space here is composed of all parallelizable loops. SP has 102 parallelizable loops, and an exhaustive search would involve $2^{102}$ runs. A tuning algorithm logically narrows down the search space by eliminating optimization variants that may have similar effects or by deriving information related to interactions between different optimizations.

The aim is to provide a generic tuning algorithm that is capable of traversing over points in the search space, irrespective of their individual characteristic contributions to the performance of the program. For example, we want to avoid considering the effects of a memory locality-based optimization on run-time properties such as cache misses, but instead, focus on the effect of the optimization on program run-time. As described above, the overall effects of serializing/parallelizing a loop are evident in the program execution time. The Combined Elimination algorithm [35] is one such generic algorithm that has previously been used to tune a generic set of compiler flags by measuring each flag’s contribution in terms of its effect on program execution time. The same algorithm is applied here to a search space composed of all the parallelizable loops in the program, each considered as a distinct on/off optimization.

To the best of our knowledge, this is the first time such a tuning algorithm is being considered to study the effects of loop parallelization on a section-by-section basis for application performance. Note that the capability of the algorithm to narrow the search space by considering interactions between different compiler flags directly translates to considering interactions between parallelized or serialized combinations of loops. Interactions make the development of a tuning method that operates on a
section-by-section basis non-trivial. However, the implementation does not rely on any compile-time decisions to improve handling of these interactions and solely allows the tuning algorithm to consider them during its traversal. This, as shown in Section 4.2, largely improves the measurement of effects of parallelization on performance.

4.1.3 Tuner implementation in Cetus

Parallelization output is generated by Cetus in the form of `pragma cetus parallel` annotations that are attached to corresponding loops in the HIR. Traversal of the HIR, specifically annotations, is used to obtain information regarding parallelized loops, and build the search space of optimizations for the tuner. Also, the annotations provide a convenient way to implement fine-grained tuning on a loop-by-loop basis. This is facilitated by selective generation of OpenMP annotations in the output pass.

Initially, relevant information is extracted during the parallelization phase to generate the tuning search space. At every iteration of the combined elimination algorithm, the next parallelization configuration (a combination of optimization variants i.e. a combination of parallel loops) is provided to the version generating output pass of Cetus via command line. The lexical order of appearance of a loop in the source program corresponds directly to its index in the optimization vector provided to the version generator. Different combinations of parallel loops can easily be considered for tuning by looking at Cetus-parallelized loops or loops that were OpenMP parallel in the input source itself. Figure 4.2 shows an example with the corresponding optimization variants that might be tested during tuning.

4.1.4 System Workflow

Figure 4.3 describes the implementation of the entire system. It is currently implemented in Python. There is no user intervention at any stage; the system is
int foo(void)
{
    #pragma cetus parallel
    #pragma cetus private(i,t)
    for ( i=0; i<50; ++i )
    {
        t = a[i];
        a[i+50] = t + (a[i+50]+b[i])/2.0;
    }
    #pragma cetus private(i,j,t)
    for ( i=0; i<50; ++i )
    {
        a[i] = c;
        #pragma cetus private(j,t)
        #pragma cetus parallel
        for ( j=0; j<100; j++ )
        {
            t = a[i-1];
            b[j] = (t*b[j])/2.0;
        }
    }
    return 0;
}

(a) (b)

Fig. 4.2. Parallelized source code (a) and the different variants of loop parallelization possible (b). (a) shows the output obtained from Cetus after loop parallelization. The tuner uses the number of parallel loops as its search space and then navigates this space using Combined Elimination to obtain the final binary vector configuration. Cetus’ output pass is used to parse the commandline vector shown in (b) and generate OpenMP directives corresponding to the input. The entire process is automatic.
completely automated. The source program is provided to the tuning framework. The first step is auto-parallelization and identification of the number of outermost parallel loops identified by Cetus. Parallelization is performed only once and the Cetus-annotated output is stored for future reference.

A backend OpenMP compiler generates the parallel code for execution and program performance is measured at runtime. We use Intel’s ICC compiler for code generation.
We first measure serial execution time and also execution time of the manually parallelized code, if available. The system measures performance using the UNIX `time` function for whole program execution; this is accurate enough for feeding back into the tuning algorithm in order to obtain the next configuration point.

The tuning process starts with a fixed initial configuration point. The optimization configuration is provided to Cetus, this time only for version generation. As shown in the example in Figure 4.2, OpenMP output code is generated for the current configuration of loops. Empirical evaluation returns execution time for different combinations of each base case. At every iteration for the current base case, Combined Elimination uses a decision-making criterion to provide the next configuration point. When the convergence criterion for the algorithm is met, a final trained configuration of parallel and serial loops is obtained.

4.2 Evaluation

This section evaluates the final outcome of the automatic tuning approach and compares the results specifically with the performance of the input source code. The aim is to consider the effectiveness of automatic parallelization and tuning in equaling or improving on the performance of input source code. Two tuning scenarios are considered: tuning automatically parallelized source code with no input parallelization information and tuning automatically parallelized code that is extracted in addition to the parallelism already present in the input source code. A third scenario of using the framework to tune only hand-parallel loops was considered, but no results are presented for this case, as the performance did not show significant improvements. A subset of benchmarks from the NAS Parallel benchmark suite and the SPEC OMP2001 suite was considered for evaluation.

We ran the tuning experiments in single-user mode on a Dual 2.33 GHz quad-core IntelE5410 system with 32GB memory per node. All parallel runs used the available 8 threads during execution i.e. all parallel loops executed under the default OpenMP
static scheduling model. Backend OpenMP code was generated using ICC’s default optimization configuration(-O2).

We conducted the initial training of all benchmarks using a small data set. Using the training input, we obtained a configuration of loops to be parallelized/serialized. For the NAS Parallel benchmarks, the training input was the W class data set. For the SPECOMP 2001 benchmarks, we used the train data sets. Our system then applied this configuration and measured parallel performance on other data sets.

Figure 4.4, Figure 4.5, Figure 4.7 and Figure 4.8 show the results of training and testing tuned configurations using different input data sets. All performance results are normalized to serial execution time. HandParallel shows the performance of manually parallelized benchmarks. CetusParallel is the performance of automatically parallelized code without any tuning or performance estimation. Cetuned is the performance obtained using our combined Cetus and tuner approach i.e. Cetune. We only tune parallelization of Cetus-parallelized loops in this case. Hand+Cetuned is the performance of the benchmarks where manually parallelized code is left untouched, while additional parallelism identified by Cetus is tuned for better performance.

4.2.1 Tuning Auto-Parallelized Code

This experiment represents the first scenario of automatic tuning. Only Cetus-parallelized loops were considered during parallel execution. All OpenMP directives in the input were removed. We were then dealing with programs that only consisted of Cetus-identified loop-level parallelism. For the starting point in the tuning search space, parallelization is turned on for all parallelizable loops in the program.

Consider the training runs in Figure 4.4 and Figure 4.5 i.e. the results obtained while training the NAS Parallel benchmarks on the W data set and the SPECOMP OMP2001 benchmarks on the train data set. It is clear that the CetusParallel version degrades performance significantly, even compared to serial execution. This can be explained by factors described at the beginning of Section 4. In a large number of
Fig. 4.4. Cetune: Tuning results for BT, CG, EP, FT and LU benchmarks. Training performed with W data set for NAS Parallel suite and train data set for SPEC OMP2001 suite. CetusParallel shows performance for compiler-parallelized code. Cetuned tunes only Cetus-parallelized loops and aims to perform close to equal or better than Serial.
Fig. 4.5. **Cetune**: Tuning results for *MG*, *SP*, *equake* and *art* benchmarks. Training performed with W data set for NAS Parallel suite and *train* data set for SPEC OMP2001 suite. *CetusParallel* shows performance for compiler-parallelized code. *Cetuned* tunes only Cetus-parallelized loops and aims to perform close to equal or better than *Serial*. 
cases, parallelization of inner loops leads to significant fork/join overheads and hence, degradation in performance. For example, in the case of LU, Cetus identifies 43 parallel loops, 22 of which are not hand-parallelized. Some of these Cetus-parallelized loops are at the 3rd or 4th nesting level which can lead to an explosion of fork/join overheads. The same can be said for BT, where 72 loops are parallelized by Cetus. Some of these are repeatedly executed inside other for loops, thus increasing parallelization overheads. For data class A, the Cetus-parallelized version of BT executes for 1555s as opposed to only 186s for the serial version. For data class B, this ratio is 6640s to 840s, thus barely showing up on the speedup chart in Figure 4.4.

Even though our tuning process starts from the configuration that yields this degradation, it effectively trains itself to a configuration that nearly equals or improves over serial execution time. For example, for the above mentioned BT results, the Cetune framework filters out loops and brings execution time to 155s for class A and 705s for class B i.e. better than serial.

For data classes A and B and the test data set for SPECOMP, the average improvement in execution time over serial execution was 35.6% with almost 3.5x increase on data class A and 2.5x increase on data class B for CG, and almost 2x increase for SP on data class A. In CG, Cetus parallelizes 16 loops as compared to the hand-parallel version, which parallelizes 24 loops. Out of the 16 Cetus-parallelized loops, only one loop does not exist in the hand-parallelized set, but nevertheless is nested inside a hand-parallel loop. The auto-parallelized version provides great speedup, almost on par with the hand-parallelized version. The tuning framework correctly retains this performance, and that too, by eliminating that single loop mentioned above, and thus provides performance equal to hand-parallelization only through 15 parallel loops. The results for SP are even more interesting. Cetus parallelizes 102 loops in SP. Only 13 of these 102 loops are serialized in the final configuration, mostly loops with small iteration counts and small loop bodies. 2 of the serialized loops were hand-parallel, adverse effects of which are seen on data class A, where the tuned version is only slightly slower than the hand-parallel version. On data class B, however,
the tuned version performs better than the hand-parallel version. Most importantly, 28 of the 102 loops were also hand-parallel and 26 of these were parallelized in the tuned version. Another 36 of the 102 loops enclosed hand-parallel loops, thus providing coarser-grain parallelism. This leads to much better performance using automatic parallelization for \textit{SP}.

We observe a slight anomaly on some benchmarks, where the final trained configuration is poorer than serial. This could easily be avoided by setting the final configuration to serial. However, it is important to notice that the trained configuration improves performance on the test data sets in most cases. This proves that Cetus correctly identifies parallel loops that are eventually beneficial for performance, even though it is not apparent during the training phase due to the short execution time for the train data sets. \textit{BT} and \textit{MG} are examples of this observation.

\textbf{Limitations: Convergence to local minima}

![Fig. 4.6. Tuning results for FT and art benchmarks by using a different start point for the search space. Training performed with A data set for FT and train data set for art. \textit{CetusParallel} shows performance for compiler-parallelized code. \textit{Cetuned} tunes only Cetus-parallelized loops and performs close to equal or better than \textit{Serial}. \textit{HandParallel} shows performance of manual parallelization.}

In the case of \textit{FT} and \textit{art}, the tuned version performs poorer than serial as shown in Figure 4.4. In spite of the improvement over \textit{CetusParallel}, the training data set
in both cases runs for about 2s, providing short execution time to determine a tuned configuration. The final configuration provides some parallel loops and promises to perform close to serial. When applied to the test data set, the degradation increases i.e. the loops identified as being possibly beneficial for the test run, result in significant overheads. Theoretically, the algorithm should identify the serial configuration as being the most beneficial. But, we believe the algorithm converges to a local minima, thus being unable to find a loop configuration that might provide better performance. This anomaly is a function of the start point in the tuning search space and the convergence criteria.

In order to tune parallel performance for the above experiment, we begin from a point where all parallelizable loops are parallelized. The algorithm improves significantly on this configuration, but nevertheless reaches a local minimum that fails to identify the fact that serial performance is the best possible performance achievable i.e the auto-parallel loops are not beneficial.

To obtain a better tuned configuration, we modify the start point, where all Cetus-parallelized loops are turned off i.e. serialized, as opposed to switched on i.e. parallelized. In the case of FT, training is performed using data class A, as class W provided no parallel loops. A few loops were parallelized which show a small benefit when tested with data class B. In the case of art, training is performed using the train data set. No auto-parallelized loops were switched on in the tuned configuration, thus leaving the performance equal to serial. Results for this tuning experiment are shown in Figure 4.6.

The above results show that extracting beneficial parallelism is a complex function of i)the loops selected for parallelism, ii)the nesting level for these loops, iii)the interaction between parallel and serial loops and also, iv)the input data set. These factors subsequently affect the search space traversal and hence bring up issues such as local minima and start configurations for tuning. As described in Section 2, model-based techniques can be used to make decisions regarding the starting point in the search space, thus playing a complementary role to empirical tuning.
4.2.2 Tuning Additional Parallelism in Hand-Parallelized Code

This experiment represents the second scenario. Only Cetus-parallelized loops that were not already parallel on input or were not enclosed by loops that were parallel on input were considered in this case. In cases where Cetus identified outer-level parallel loops as opposed to those parallelized in the input, the Cetus-identified level was parallelized during evaluation. For the starting point in the tuning search space, parallelization was turned off for all additional parallelizable loops in the program.

Figure 4.7 and Figure 4.8 show that the training phase of tuning for additional parallel loops yields little performance benefit in most cases. For EP, the training phase shows slight degradation, this is due to short execution time e.g. the Hand version runs 0.7s on the W data set while the Hand+Cetuned version runs for 0.8s. On using the parallelized set for the production runs, we see that the Hand+Cetuned version performs equal to or better than Hand. On the other hand, SP shows speedup for Hand+Cetuned even during the training phase. Training was however performed on a small data set. The final tuned configuration did show some additional auto-parallelized loops during the training phase in almost all benchmarks. These configurations were then applied to the reference data sets. In most cases, the benchmarks still behaved the same i.e. the tuning of additional parallel loops yielded little performance increase for data classes A and B. In the case of CG, data class A shows degradation in the Hand+Cetuned case, but this is misleading because the parallel tuned version executes only for 0.86s, as compared to the Hand version that executes for 0.82s. These small differences are ironed out in the results for data class B.

Automatic parallelization yields significant benefits in the case of SP, where we find parallelism at outer levels in the loop nests that are manually parallelized at inner nesting levels. Cetus parallelizes 74 additional loops, out of which 25 loops are parallelized by the final trained configuration. Out of these 25, 12 loops enclose originally hand-parallel loops and thus provide coarser-grain parallelism. Hand+Cetuned identifies coarser-grain parallelism as beneficial in cases where it is proving to be ben-
Fig. 4.7. **Hand+Cetune**: Tuning results for BT, CG, EP, FT and LU benchmarks. Training performed with W data set for NAS Parallel suite and train data set for SPEC OMP2001 suite. **Hand+Cetuned** tunes parallel loops that exist in addition to hand-parallel loops **Hand-Parallel**.
Fig. 4.8. **Hand+Cetune**: Tuning results for *MG, SP, equake* and *art* benchmarks. Training performed with W data set for NAS Parallel suite and *train* data set for SPEC OMP2001 suite. *Hand+Cetuned* tunes parallel loops that exist in addition to hand-parallel loops *Hand-Parallel*. 
eficial. In many cases, the outer parallel loop has a very low iteration count, and hence the inner hand-parallel loop is the level defined as beneficial in the final configuration. Our Hand+Cetuned SP version performs almost 39% better for class A and almost 23% better for class B than manual-parallelization.

4.2.3 Tuning time

Tuning time was in the order of minutes for training each of the 9 benchmarks and eventually obtaining the best possible tuned loop configurations. Having trained on small data sets, the reference data runs would require only three runs to measure relative performance benefits of automatically parallelized and automatically tuned code, which would require time corresponding to a particular data set for the given application.

4.2.4 Intel’s ICC Auto-Parallelizer

We use ICC’s automatic-parallelizer to study a state-of-the-art approach based on compile-time heuristics to parallelize loops. We tested the parallelizer using three different parallelizing thresholds, which can be user-defined. The auto-parallelizer uses the threshold (0-100) as a probability measure of the chances of performance improvement as determined through compile time heuristics. The default threshold of 100 means that ICC parallelizes the loop only if it is 100% sure that parallelization would yield performance benefit. This is only possible if it knows loop iteration counts at compile time. Hence, we also tested using a threshold of 99 which implies all loops considered 99% beneficial but not certainly beneficial can be parallelized. Lastly, a threshold of 0 was used, which tells ICC to parallelize all loops irrespective of performance considerations.

ICC performed extremely poorly at threshold 0 on equake, BT, LU, MG with upto 90% degradation in performance. The behavior at threshold 100 was inconsistent. In almost all cases, the parallelized version performed only as well as the serial
execution. In the case of \textit{LU}, the performance at threshold 100 was down 65\% from serial execution. Similar results were observed at threshold 99.

The results highlight the inadequacy and inconsistency of a purely compile-time approach to automatic parallelization. Using the Cetus parallelizer and an empirical tuning approach, we obtain significantly better application performance.

4.2.5 Dynamic Tuning Considerations

An important observation with regards to several of the benchmarks is that the hand-parallelized version itself shows little speedup for the \textit{train} data set. However, for \textit{art}, we get almost 4x speedup for the \textit{ref} data set. In some other cases, the \textit{train} data shows more speed up, but reduced or varying speedup for the \textit{test} data sets e.g. in the case of \textit{BT, CG, MG} and \textit{SP}. This points to the fact that the parallelism is highly input-dependent and that, in such a case, off-line tuning using a different data set may yield little to no benefit for other data sets. Dynamic tuning would, on the other hand, be able to take input-dependent effects into consideration. However, dynamic tuning comes at the cost of incurring runtime overheads. Also, self-adapting code is unfavorable in many cases, owing to correctness and security considerations. Moreover, widely accepted techniques such as profiling for performance data are highly input dependent but are known to perform well. This argument favors our current offline tuning approach, which works well for the set of benchmarks being considered.

In the case of \textit{art}, the \textit{train} data set offers little opportunity to recognize beneficial parallelism. Despite this fact, the tuner identifies loops that are potentially beneficial and this is evident in the results for the \textit{ref} data set where the tuned version improves significantly over the automatically parallelized version.
5. CONCLUSIONS

The research described in this thesis has considered the foundational techniques of automated parallelization and performance tuning, which will play an important role in the development of parallel software in the multicore era. The scope of parallelism is widening every day, and the onus of developing software with sufficient parallelism, performance and portability is entirely on the program developer. Automatic parallelization is an important technique by which the compiler is able to perform complex analyses on the program and automatically extract parallel sections of code. Such a process can significantly cut down development time. A wide variety of new approaches are necessary to meet the demands of parallelization, and help in the process of creating good parallel software. Just as with different programming languages in the serial domain, the variety of approaches and programming models in the parallel domain is unavoidable. Automatic parallelization can help in achieving parallelization across different platforms and thus, improve portability as well.

This thesis has described a subset of source-to-source parallelization capabilities implemented in the Cetus compiler infrastructure. Data dependence analysis and loop parallelization are key steps in identifying loop-level parallelism and in expressing this parallelism in the OpenMP shared memory parallel programming model. For parallelizing a language like C, handling pointers and aliases is crucial not only for correctness, but also for enabling other improved analyses and thus helping the compiler generate better code. This is made possible by the points-to analyzer implemented in Cetus. The intraprocedural points-to analysis forms a major part of this analysis and is described in this research. Parallelization results for the NAS Parallel benchmark and SPEC OMP2001 benchmark suites show that Cetus can extract a significant number of parallel loops. Through advanced alias analysis, the number
of parallel loops is increased even further and a significant proportion of these are hand-parallelized loops as well.

Auto-tuning research considers the important issue of finding profitable parallelism for auto-parallelization of both serial and parallel programs. This research describes the implementation of an automatic parallelizer that nearly equals or improves in performance over the original program execution across a large set of benchmarks. The system applies an offline tuning approach that has previously been used to tune generic compiler flags. We have adapted the algorithm to tune the parallelization of loops in the program and to apply this optimization on a loop-by-loop basis. We have shown that the programs, tuned with a training data set, always perform near-equal or better than the original code on the production data sets.

We have provided results for the implementation on a subset of the NAS Parallel and SPEC OMP2001 benchmarks. Our auto-parallelized and auto-tuned benchmarks perform 35.6% better than serial on average. Tuning additional parallelism in hand-parallelized codes performs equal to or better than the original programs (20-40% better in the case of SP).

While overzealous parallelization of loops is the biggest source of overheads in auto-parallelized programs, the tuning system can be extended to other optimization techniques that are available in the Cetus infrastructure. Other future work can also consider combining empirical tuning with model-based approaches, so as to reduce tuning time as much as possible. Furthermore, for large programs, offline tuning can be expected to face limits, as different data sets may exhibit significantly different execution paths through the program. For this case, dynamic strategies can be developed so that they can tune during production runs of an application, without the need for a training phase.
LIST OF REFERENCES


