SCOOP
Language extensions and compiler optimizations for task-based programming models

Foivos S. Zakkak

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. *Angelos Bilas*

This work has been performed at the Foundation for Research and Technology Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Av., Vassilika Vouton, Herakleion, GR-70013, Greece.

The work is partially supported by the 7th European Commission Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement no 248647, and the HiPEAC1 and HiPEAC2 Networks of Excellence, grant agreement no 004408 and no 217068 respectively

This work made use of the facilities of HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme.

University of Crete
Computer Science Department

SCOOP: Language extensions and compiler optimizations for task-based programming models

Thesis submitted by
Foivos S. Zakkak
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Foivos S. Zakkak

Committee approvals: _____
Angelos Bilas, Professor, Thesis Supervisor
Computer Science Department
University of Crete

_____
Dimitrios S. Nikolopoulos, Professor
School of Electronics, Electrical Engineering and Computer Science
Queen's University of Belfast

_____
Evangelos Markatos, Professor
Computer Science Department
University of Crete

Departmental approval: _____
Angelos Bilas, Professor
Director of Graduate Studies

Heraklion, 28 February 2012

Abstract

Over the past decade, CPU development has focused mainly on multi-core architectures, and recent trends lead to multi-core designs with ever increasing numbers of cores. In order to get the best out of a many-core system, one has to develop efficient parallel applications. However, reasoning about synchronization, ordering and conflicting memory accesses makes parallel programming difficult, error-prone and hard to test, debug and maintain. Task-based programming models such as OpenMP, Cilk, Sequoia. SvS, OoOJava and StarSs offer a more structured way of expressing parallelism than threads, while some of them are able to automatically infer parallelism and dependencies.

Despite their advantages, task-based programming models impose significant challenges for the runtime system. Fine-grained tasks require efficient basic mechanisms for task management. Task management operations, such as initiation, completion, queuing, and scheduling, in traditional parallel systems cost in the order of tens of thousands of cycles, due to communication and memory management overheads.

In this thesis, we present SCOOP (**S**ource-level **CO**mpiler **O**ptimizations for **P**arallelism), a C source-to-source compiler for task-based programming models that use dataflow annotations. SCOOP extends C with dataflow annotations using `#pragma` directives, like StarSs. SCOOP parses these annotations and produces efficient C source code for task-parallel runtimes. SCOOP is also capable to infer independent task arguments, enabling it to hint either the programmer or runtimes that implement some kind of dynamic dependence analysis. We evaluate SCOOP's effectiveness on both *symmetric multiprocessing (SMP)* and the *Cell Broadband Engine* architectures ,using a set of parallel benchmarks.

Περίληψη

Κατά την τελευταία δεκαετία, η ανάπτυξη κεντρικών μονάδων επεξεργασίας έχει επικεντρωθεί κυρίως σε πολυπύρηνες αρχιτεκτονικές, και οι πρόσφατες τάσεις οδηγούν σε πολυπύρηνους επεξεργαστές με όλο και μεγαλύτερο αριθμό πυρήνων. Για να εκμεταλλευτεί κανείς πλήρως ένα πολυπύρηνο σύστημα, πρέπει να αναπτύξει αποδοτικές παράλληλες εφαρμογές. Ωστόσο, η συλλογιστική σχετικά με το συγχρονισμό, την σειρά εκτέλεσης και τις συγκρουόμενες προσβάσεις στην μνήμη καθιστά τον παράλληλο προγραμματισμό περίπλοκο, επιρρεπή σε λάθη και δύσκολο να δοκιμασθεί, να διορθωθεί και να διατηρηθεί. Τα μοντέλα task-based προγραμματισμού όπως τα OpenMP, Cilk, Sequoia. SVS, OoOJava και StarSs προσφέρουν έναν πιο δομημένο τρόπο έκφρασης της παραλληλίας από ότι τα threads, ενώ κάποια από αυτά είναι σε θέση να εξάγουν αυτόματα παραλληλισμό και να επιλύουν εξαρτήσεις.

Παρά τα πλεονεκτήματά τους, τα task-based μοντέλα προγραμματισμού προκαλούν σημαντικές προκλήσεις για το σύστημα εκτέλεσης (runtime). Τα tasks που είναι fine-grained απαιτούν αποτελεσματικούς βασικούς μηχανισμούς για τη διαχείρισή τους. Όμως οι μηχανισμοί διαχείρισης tasks, όπως η αρχικοποίηση, η ολοκλήρωση, η τοποθέτηση σε λίστες αναμονής, και ο χρονοπρογραμματισμός, σε παραδοσιακά παράλληλα συστήματα έχουν κόστος της τάξης των δεκάδων χιλιάδων κύκλων, που οφείλεται στην επικοινωνία και τη διαχείριση μνήμης.

Στην παρούσα εργασία, παρουσιάζουμε τον SCOOP (**S**ource-level **CO**mpiler **O**ptimizations for **P**arallelism), έναν C source-to-source μεταγλωττιστή για task-based μοντέλα προγραμματισμού που χρησιμοποιούν επισημειώσεις ροής δεδομένων (dataflow annotations). Ο SCOOP επεκτείνει την C με επισημειώσεις ροής δεδομένων χρησιμοποιώντας οδηγίες τύπου `#pragma`, όπως αυτές του StarSs. Ο SCOOP αναλύει αυτές τις επισημειώσεις και παράγει αποδοτικό κώδικα C για task-parallel runtimes. Ο SCOOP είναι επίσης σε θέση να συμπεράνει αν κάποιες παράμετροι των task είναι ανεξάρτητες, ώστε να βοηθήσει είτε τον προγραμματιστή ή τα ρυντιμες που εφαρμόζουν κάποιο είδος δυναμικής ανάλυσης εξαρτήσεων. Τέλος, αξιολογούμε την αποτελεσματικότητά του SCOOP σε επεξεργαστές αρχιτεκτονικών *symmetric multiprocessing (SMP)* και *Cell Broadband Engine*, χρησιμοποιώντας ένα σύνολο παράλληλων προγραμμάτων, ως ορόσημα (benchmarks).

Acknowledgments

I would like to express my sincere acknowledgment to my supervisor and academic advisor, professor Angelos Bilas, professor Dimitris S. Nikolopoulos and post-doc researcher Polyvios Pratikakis. Their invaluable assistance, support and guidance gave me a better understanding of Parallel and Distributed Systems, enabling me to develop a new perspective for Parallelism.

I would also like to show my gratitude to Dimitris Chasapis and George Tzenakis , research assistants of the CARV laboratory of ICS-FORTH. I am also grateful towards ICS-FORTH, for the provided student scholarship.

Finally i would like to thank my family, especially my sister Sylvia for reviewing the text, and my roommate Christos Moschovitis for their support during my studies.

Contents

List of Figures

List of Codes

List of Tables

Chapter 1

Introduction

In this thesis, we present SCOOP (**S**ource-level **CO**mpiler **O**ptimizations for **P**arallelism), a C source-to-source compiler for task-based programming models that use dataflow annotations. Also we present an extension to task-based programming models, region support. SCOOP extends C with dataflow annotations using `#pragma` directives. SCOOP parses these annotations and produces C source code for task-parallel runtimes. SCOOP is capable to infer independent arguments using the *Static Dependence Analysis Module*, which can be used to hint either the programmer or runtimes that implement some kind of dynamic dependence analysis.

1.1 Motivation

Parallel programming involves mapping computations that can be done at the same time onto many processing elements, as well as defining how these elements interact and communicate.

The two main, explicitly parallel programming models used today are shared memory and message passing. Shared memory requires programs to specify synchronization information for memory accesses. Message passing on the other hand requires programs to deal with data placement and communication buffer management. Reasoning about the implicit thread interactions through shared memory and manually controlling all synchronization, as well as sending the appropriate data and manually managing the buffer is difficult and error prone, resulting in data races, deadlocks and other concurrency errors that are difficult to reproduce and fix, due to the inherent nondeterminism of the two main parallel programming models.

Early task-parallel programming models, such as Cilk [1] and OpenMP [2] raise the level of abstraction for expressing parallelism, but still require explicit synchronization. Statically prohibiting concurrent access to shared memory among parallel tasks is too restrictive, as in many cases only a few instances of conflicting tasks will actually conflict. For this reason, even existing statically verifiable,

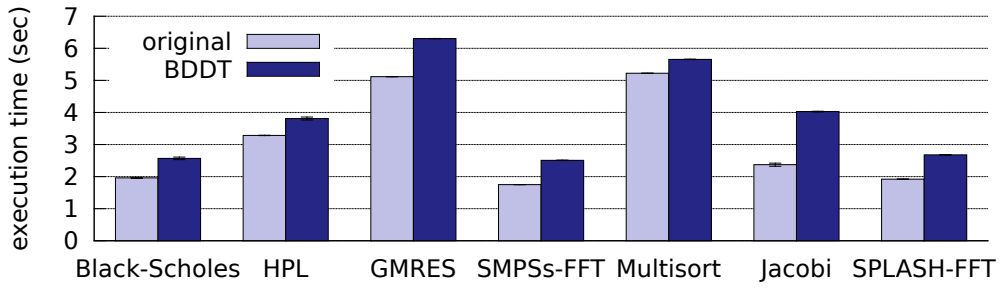


Figure 1.1: Overhead of running BDDT on a single core

non-dependent task models relax the requirement for strictly non-overlapping task memory footprints. Instead, recent task-based programming models implement implicit synchronization, using the memory footprint of every task, either inferred by the system or declared by the programmer, to avoid concurrent accesses or achieve fully deterministic execution [3, 4, 5, 6, 7].

Despite their advantages, task-based programming models impose significant challenges for the runtime system. Fine-grained tasks require efficient basic mechanisms for task management, as task initiation and completion now become common-path operations. Task management operations, such as initiation, completion, queuing, and scheduling, in traditional parallel systems costs are in the order of tens of thousands of cycles, due to communication and memory management overheads.

Furthermore, dynamic dependence analysis incurs a high overhead compared to hand-crafted synchronization; It requires a complex runtime system to manage and track memory allocation, check for conflicts, and schedule parallel tasks. Often, the runtime cost of checking for conflicts in pessimistic, or rolling back a task in optimistic runtimes becomes itself a bottleneck, limiting the achievable speedup as the processor count grows. Figure 1.1 compares the BDDT [6] runtime dependence analysis when run on a single processor with running the sequential program, for a set of benchmarks. On average, BDDT incurs an overhead of 33%. Along the same lines, Best et al. [5] report overhead costs ranging from under 5% to over 40% for the dynamic dependence analysis in SvS.

We believe that these overheads could be reduced by statically transforming the applications source code and generating custom task management code to interact with the runtime, while hinting it of arguments that can be ignored by the dynamic dependence analysis.

1.2 Background

1.2.1 Runtimes

SCOOP is designed to generate code for task-based programming models. The

notion of a task is general and can be interpreted in various ways. In our work we consider a task to be a piece of code that can execute in parallel with other tasks, *as well as* the data that it will access. SCOOP currently generates code for two task-based programming models, TPC [8] and BDDT. In this work we focus on the later as SCOOP code generation for the TPC runtime has been examined previously in Foivos S. Zakkak's BSc thesis [9].'

We will now briefly discuss BDDT. BDDT is a task-parallel runtime system that dynamically discovers and resolves task dependencies. SCOOP compiles task-parallel programs to use the BDDT runtime for detecting dependencies and scheduling parallel tasks deterministically. BDDT requires the programmer to specify the memory footprint of every task as a set of input, output, or input-output effects on memory address ranges or multidimensional array tiles. BDDT then uses a block-based dependence analysis at customizable granularity, to detect possible overlap in task footprints and enforces sequential program order on conflicting tasks.

BDDT uses a custom slab memory allocator to create a metadata element for each allocated memory block. Due to the allocation policy, metadata elements are accessible at constant time for every arbitrary memory address. Each metadata element describes the most recent state of the memory block, tracking its version, which changes every time the block is written. This allows for many-readers/one-writer synchronization, equivalent to read-write locking. Metadata elements are internally protected by fine-grain locks, as many threads may update the same metadata concurrently when scheduling tasks. Using per-block metadata allocated in the slab-allocator allows BDDT to schedule tasks operating at arbitrary memory locations, common in a language with unrestricted pointer arithmetic such as C. In addition, BDDT maintains per-task metadata, including

1. an atomic counter that tracks the number of unresolved dependencies
2. links to metadata elements describing all memory blocks in the task footprint
3. the task's effect (read, write, read/write) on the data

Using the task and memory metadata, BDDT can detect task dependencies and enforce sequential program order among conflicting tasks. Deciding if a memory block causes a dependence ,has constant $O(1)$ cost, and scheduling a task incurs cost linear to the number of contiguous blocks in the task footprint. This makes BDDT perform better than OpenMP in many programs, as dynamic dependence analysis enables more parallelism than synchronization with barriers and locks. However, although BDDT has been carefully optimized, this overhead becomes significant in programs with fine-grain independent tasks, where dynamic checks are always unnecessary, therefore limiting scalability and parallelism.

1.2.2 SDAM

Static **D**ependence **A**nalysis **M**odule [10] is a SCOOP extension that finds whether task arguments should be treated as potential dependencies between tasks. To

achieve this, it employees Locksmith's [11] Points-to analysis, which implements a type-based flow analysis to detect pointer aliasing. This method is a may alias analysis, meaning that we cannot decide whether two pointers alias for sure, but we can decide whether they don't refer to the same memory address/variable. In the later case, we can safely deduct that two arguments are independent. To make the analysis more accurate, it uses the programmer's dataflow annotations, and allows concurrent reads.

1.2.3 Regions

Trends show that programs' needs of space grow along with computer's memory. Furthermore, many programs manage to totally allocate memory that exceeds the size of the physical memory, sometimes resulting in memory swap, a rather slow procedure. On early work in programming languages, stack-based memory management was employed to address this problem. However there are still some limitations related to dynamic structures, such as lists, hashtables etc. For stack-based memory management to work, the size of a value must be known at the beginning of a block structure, where the stack allocation is performed. Moreover stack-based memory management requires that the lifetime of values stored in the stack-allocated memory complies with the start and the end of the block structure. However this is not always the case, many times the programmer wants some data to stay alive even after the end of the block structure. In such cases the programmer uses `malloc` and `free` to explicitly manage the memory allocation. These techniques require the programmer to know exactly when a block of memory does not contain live variables and usually result in memory leaks.

Region-based memory management is a form of compile-time memory management, well-known from the functional programming world. Region-based memory management [12] is something between completely manual and completely automatic memory management. We use an abstraction of this concept. In this work each region is like a stack of unbounded size which grows, until the region in its entirety is popped off the region stack.

With regions support in task-based programming models we can:

1. express complex task footprints
2. dynamically allocate or deallocate memory within tasks
3. reduce memory management overhead
4. reduce dependence analysis overhead
5. increase memory locality

1.3 Related Work

1.3.1 Task parallelism

There are several programming models and runtime systems that support task parallelism. We will now examine CellSs [13], SMPSs [14], OpenMP [2], Thread Building Blocks [15], Cilk [1], and Sequoia [16].

Cell Superscalar (CellSs) addresses the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the Cell BE architecture. The focus in on the simplicity and flexibility of the programming model. Based on a simple annotation of the source code, a source to source compiler generates the necessary code and a runtime library exploits the existing parallelism by building at runtime a task dependency graph. The runtime takes care of the task scheduling and data handling between the different processors of this heterogeneous architecture. In comparison, SCOOP performs custom code generation and can hint the runtime about independent arguments, optimizing the resulting code, while CellSs' source to source compiler only generates the appropriate runtime calls.

SMP superscalar (SMPSs) is a task-parallel programming model focused on the ease of programming, portability and flexibility that is based on CellSs. Contrary to CellSs, SMPSs is tailored multi-cores and Symmetric Multiprocessors (SMP) in general. The runtime takes care of scheduling the tasks and handling the associated data. The same differences as in CellSs apply here too.

OpenMP is a set of compiler directives for parallelizing sequential code, implemented using a standardized API. OpenMP expresses shared memory loop and task-parallelism, and supports specific directives for specifying local and shared memory accesses used to optimize the code. However, it does not enforce any synchronization automatically, requiring the programmer to manually use barriers, locks, or other synchronization to avoid concurrency errors. OpenMP in comparison with SCOOP doesn't perform any source-to-source transformations. Contrary it is implemented in the compiler itself, producing directly the appropriate assembly code. OpenMP's standardized API, reduces its flexibility and prevents it from evolving with the current trends. On the other hand, SCOOP's syntax can easily evolve with the current programming needs, and SCOOP itself can be ported to many platforms, by adding new code generation modules.

Cilk/Cilk++ is a parallel programming language that extends C++ with recursive task-based parallelism for shared memory systems. Cilk expresses parallelism using a *spawn* statement to state that a function invocation can be computed in parallel, and synchronization using the *sync* statement to state that a parent task should wait until all its spawned children have finished. Cilk tasks can be very

fine-grained without incurring much overhead, because Cilk only executes *spawn*s in parallel if necessary, using a work-stealing scheduler. Again, the programmer is responsible to use *sync* or any other synchronization mechanism to avoid data races and enforce specific task orderings. Cilk uses a dynamic analysis to detect nondeterminism [17].

Sequoia is a parallel C++-like programming language that can target both shared memory and distributed systems. In Sequoia, the programmer writes a hierarchy of nested parallel tasks, where leafs are atomic and perform simple computations, and inner tasks break down the computation into smaller sub-tasks, and combine their results; a machine description that specifies the various levels in the memory hierarchy, whether memory is shared, etc.; and a mapping file that describes how data is broken and distributed among tasks and their sub-tasks, which tasks are scheduled to run over which level of the memory hierarchy, and when computation workload should be broken into smaller tasks. In comparison, SCOOP and BDDT require no task nesting hierarchy specification, machine description nor mapping file. With SCOOP the programmer annotates the function calls, she would like to be executed in parallel, with `#pragma` directives. SCOOP then generates the appropriate code to be linked with the BDDT's library.

All these programming models except SMPSs and CellSs do not provide implicit synchronization. In the next subsection we discuss such programming models and languages including SMPSs and CellSs.

1.3.2 Implicit Synchronization

Several programming models and languages aim to automatically infer synchronization among parallel sections of code. Transactional Memory [18] preserves the atomicity of parallel tasks, or transactions, by detecting and retrying any conflicting code. Static lock allocation [19] provides the same serializability guarantees by automatically inferring locks for atomic sections of code. These attempts, however, allow nondeterministic parallel executions, as they only enforce race freedom or serializability, not ordering constraints among parallel tasks.

Recently, task-parallel models perform static and dynamic dependence analysis to detect task dependencies, using the task's memory footprint. BDDT is a runtime system that detects and enforces task dependencies at the memory block level. BDDT requires the programmer to provide the task footprint at arbitrary address granularity, can express footprints of multidimensional array tiles, and incurs little memory overhead. The scalability in such systems is limited by the runtime overhead for calculating task dependencies. SCOOP compiles a task-parallel program to use the BDDT runtime and uses static analysis to reduce that runtime cost. Not counting the dependence analysis, BDDT has been shown to perform similarly or better to OpenMP.

StreamIt [20] implements a dataflow model of computation where the program includes static dependencies among the various stages of the pipeline. Since all dependencies among parallel code sections are explicit, the compiler generates all synchronization, data transfer and copying necessary to guarantee deterministic execution that preserves the ordering among parallel tasks.

SMPSs and CellSs are two more runtime systems that use dynamic analysis to detect dependencies between tasks, focusing on scientific programs with arrays, written in the StarSs programming model. In comparison, SCOOP and BDDT support tasks operating on multidimensional array tiles with arbitrary overlap, and use a context-sensitive static dependency analysis to avoid runtime overheads unless necessary.

SvS [5] uses a custom task-description language, and a static dependence analysis that determines all reachable objects for each task. A runtime system then uses an efficient approximate representation of the reachable object sets, resembling Bloom filters to enforce mutual exclusion, not deterministic ordering, between conflicting tasks. Moreover, SvS object reachability sets are approximate and may include many reachable objects in the program, regardless of whether they are actually accessed by a task instance. This may hinder the available parallelism, and does not take advantage of programmer knowledge about the memory footprint of each task. In contrast, SCOOP uses a static dependency analysis to exclude possible dependencies and remove unnecessary runtime checks, it does not require a type-safe language or object granularity on task footprints and it uses the BDDT runtime to guarantee deterministic program order.

Out-of-Order Java [3] and Deterministic Parallel Java [4, 21] , task-parallel extensions of Java, use a combination of data-flow, type-based, region and effect analyses to statically detect or check the task footprints and dependencies in Java programs at object granularity. OoOJava then enforces mutual exclusion on conflicting tasks and DPJ restricts uses transactional memory to roll back tasks in case of conflict.

CommSet [22] defines commutative sets that describe sets of commutative tasks in a parallel program. The compiler can use this information to allow more possible orderings in a program and extract more parallelism. As with all compiler-based parallelization techniques, this approach is limited by overapproximation in static pointer and control flow analyses, that might cause many tasks to be run sequentially, because only two instances have clearly disjoint memory footprints. To avoid this, CommSet uses optimistic transactional memory, which, however is not suitable for programs with high contention or effects that cannot be rolled back.

1.3.3 Static Analysis

Static dependence analysis is often employed in compilers and tools that optimize existing parallel programs or for automatic parallelization. Early parallelizing compilers used loop dependence analysis to detect data parallelism in loops operating on arrays [23, 24]. These systems, however do not handle inter-loop dependencies and do not work well in the presence of pointers.

Several pointer analyses have been previously used to detect dependencies and interactions in parallel programs. Naik and Aiken [25] extend pointer analysis with *must-not-alias* analysis to detect memory accesses that cannot lead to data races. Pratikakis et al. [26, 11] use a context-sensitive pointer and effect analysis to detect memory locations accessed by many threads in the Locksmith race detector. SCOOP uses the pointer analysis in Locksmith to detect aliasing between footprints of different tasks, and extends Locksmith's flow-sensitive dataflow analysis to detect tasks that can run in parallel.

Chapter 2

Design and Implementation

We have implemented SCOOP as a source-to-source compiler that produces GCC code using CIL [27] as a C front-end and SDAM [10] and Locksmith [26] to generate and solve points-to and control-flow constraints.

We have designed the code generation module to be completely separated from the others. This way we gave SCOOP portability over different platforms. SCOOP maintainers can, with only a few changes in the main tool, write a new code generation module. The process is as simple as using a template to write the new code generation module and add any required flags, with their handlers, to the main tool.

Figure 2.1 shows the full compilation process using SCOOP. First SCOOP parses the source code files and, using SDAM and Locksmith, generates a single or two (depending on the architecture) source code files. Then this file/files is compiled using gcc, targeting the appropriate platform/architecture. Finally we link the compiled code with the runtime's library.

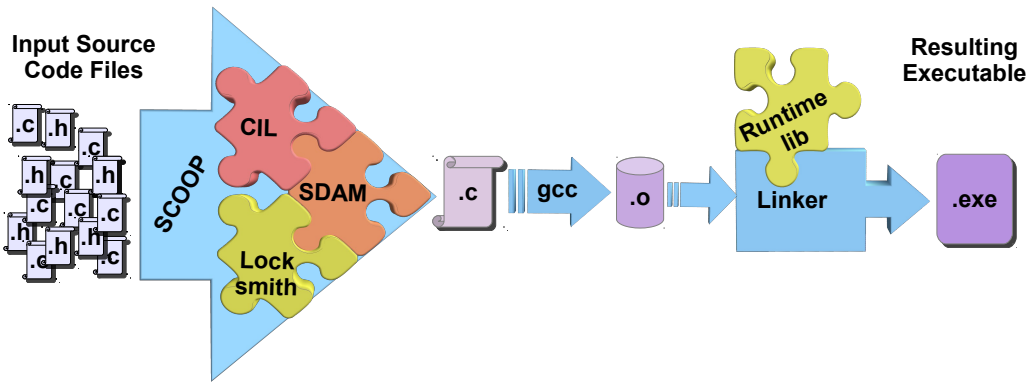


Figure 2.1: The toolchain

SCOOP is structured as three modules. In Figure 2.2 we see a graphical representation of a SCOOP pass. First SCOOP parses and merges all the source files in a single **A**bstract **S**yntax **T**ree (AST). Then it uses SDAM's static analysis to

infer independent arguments. Finally, SCOOP generates custom code that uses BDDT for creating and managing tasks, disabling BDDT's runtime dependence checks for inferred or declared independent arguments and making some further improvements.

In the rest of this chapter we examine those three modules in more detail.

Figure 2.2: The three SCOOP's modules

2.1 Syntax

The first module extends the C front-end with support for OpenMP-like `#pragma` directives to define tasks and SMPSs-like syntax to define task footprints. We have chosen to mark task creation at the calling context, instead of marking a function definition and have every invocation of the function create a parallel task as in SMPSs, for two reasons: First, this way we are able to differentiate when a function is called sequentially or asynchronously as a parallel task, and second, because depending on the arguments and calling context, an argument may or may not be safe, and fixing the type of the task footprint for all its invocations would prohibit that. The syntax for declaring task footprints supports strided memory access patterns, so that we can describe multidimensional array tiles as task arguments. When not explicitly given, we assume that the size of a task argument is the size of its type.

SCOOP supports the following `#pragma` annotations [1]

- `#pragma scoop start ( <parameters>` [2] `)`
  initializes the runtime
- `#pragma scoop finish`
  finalizes the runtime

[1] Regular expressions: * means 0 or more, + means 1 or more (comma seperated), { and } are used for grouping, | means or, ? means 0 or 1

[2] depends on the runtime's requirements

- `#pragma scoop wait all`
  implements a global barrier

- `#pragma scoop task\`
  `{ in|out|inout|safe( {<task arguments>}+ ) }*|`
  `{ region <variable name> in|out|inout( {<variable name>}+ ) }*`

  Task argument notation:
  Non stride: `<variable name>{[variable size]}?`
  Stride: `<variable name>[Block Rows|Block Columns]\`
  `[ {Array Rows [3] |}? Array Columns]`

2.1.1 Example

Code 2.1: A simple program

```
1  int a = 1;
2  int b = 2;
3  int c = 3;
4  int d = 4;
5
6  void set(int *x, int *y) { *x = *y; }
7  void addto(int *x, int *y) { *x += *y; }
8
9  void main() {
10   addto(&a, &b);
11   addto(&c, &d);
12
13   addto(&a, &c);
14
15   set(&b, &a);
16   set(&c, &a);
17   set(&d, &a);
18 }
```

Consider the C program in Code 2.1. This program has four global integer variables, `a`, `b`, `c` and `d` (lines 1–4). Function `set()` copies the value of its second argument to the first (line 6), and function `addto()` adds the value of its second argument to the value of its first (line 7). The two functions are then used to calculate the sum of the four global integer variables, and store it in each of them. To achieve this, `addto()` is invoked to add first `b` to `a`, then `d` to `c` and finally `c` to `a` (lines 10–13). Then `set()` is invoked to assign the value of `a` to `b`, `c` and `d` (lines 15–17).

[3] Array Rows is optional and is totally ignored

In this program the first two calls to `addto()` (lines 10–11) operate on different variables, and any execution order of them would give the same result, meaning they can be executed in parallel. Furthermore all three calls to `set()` (lines 15–17) write on different variables but all of them read from `a`, meaning that after `a` is written they can be executed in parallel.

At this point we can break the program in three different segments where the code of the first and the third segment can execute in parallel. Code 2.2 shows how we would rewrite the C program from Code 2.1 using SCOOP, considering that the task-parallel runtime we are targeting doesn't perform dynamic dependence analysis, thus inserting barriers to manually resolve dependencies.

Code 2.2: Code 2.1 written using SCOOP annotations

```
1  int a = 1;
2  int b = 2;
3  int c = 3;
4  int d = 4;
5
6  void set(int *x, int *y) { *x = *y; }
7  void addto(int *x, int *y) { *x += *y; }
8
9  void main() {
10   #pragma scoop start(...)
11
12   #pragma scoop task inout(&a) in(&b)
13   addto(&a, &b);
14   #pragma scoop task inout(&c) in(&d)
15   addto(&c, &d);
16
17   #pragma scoop wait all
18   addto(&a, &c);
19
20   #pragma scoop task out(&b) in(&a)
21   set(&b, &a);
22   #pragma scoop task out(&c) in(&a)
23   set(&c, &a);
24   #pragma scoop task out(&d) in(&a)
25   set(&d, &a);
26
27   #pragma scoop finish
28 }
```

In this program the first two calls to `addto()` (lines 12–15) execute in parallel. Then the program waits at line 17 until they both finish. After both tasks finish, the third call to `addto()` (line 18) is executed sequentially followed by the three

parallel executions of `set()` (lines 20–25). Finally the program waits again for all tasks to finish at line 23.

2.2 Argument Independence Inference

The second SCOOP module (SDAM) uses a type-system to generate points-to and control-flow constraints, and then uses the Locksmith engine to solve them and infer argument independence.

SCOOP treats input and output task arguments differently. In particular, we match the behavior of the BDDT runtime, which allows multiple reader tasks of a memory location to run in parallel. Thus, we also mark task arguments that are only read in parallel as independent.

To increase the analysis precision, we employ the context-sensitive, field sensitive points-to analysis of Locksmith and a context-sensitive control-flow analysis. In both cases, context sensitivity is encoded as CFL-reachability, with either points-to or control-flow edges, that enter or exit a calling context, marked as special *open* or *close parenthesis* edges [28, 29].

Finally, in several benchmarks, tasks within loops access disjoint parts of the same array. However, Locksmith's points-to analysis treats all array elements as one abstract location, producing false aliasing and causing such safe arguments to be missed. To rectify this, in part, we have implemented a simple loop-dependence analysis that discovers when different loop iterations access non-overlapping array elements. This (orthogonal) problem has been extensively studied in the past [30, 24, 31, 23], resulting in many techniques that can be applied to improve the precision of this optimization.

Code 2.3: Task parallel example program

```
1  int a = 1;
2  int b = 2;
3  int c = 3;
4  int *alias = &b;
5
6  void set(int *x, int *y) { *x = *y; }
7  void addto(int *x, int *y) { *x += *y; }
8
9  int main() {
10   #pragma scoop start(...)
11
12   #pragma scoop task inout(&b) safe(&c)
13   addto(&b, &c);
14
15   #pragma scoop task safe(&a) in(alias)
16   set(&a, alias );
17
```

```
18    #pragma scoop wait all
19
20    #pragma scoop task safe(&a) safe(&c)
21    set(&a, &c);
22
23    #pragma scoop finish
24  }
```

2.2.1 Example

Consider the C program in Code 2.3. This program has three global integer variables, `a`, `b` and `c` (lines 1–3) and a global pointer `alias` (line 4) that points to `b`. Function `set()` copies the value of its second argument to the first (line 6), and function `addto()` adds the value of its second argument to the value of its first (line 7).

The two functions are then invoked in two parallel tasks, to add `c` to `b` (lines 12–13) and to set the value of `a` to the value pointed to by `alias` (lines 15–16). The first task reads and writes its first argument, `b`, and reads from its second argument, `c`. Similarly, the second task writes to its first argument, `a`, and reads from its second argument `alias`. The program then waits at a synchronization point for the first two tasks to finish (line 18) and then spawns a third task that reads from `c` and writes to `a` (lines 20–21).

To execute this program preserving the sequential semantics, the second task `set()` needs to wait until the value of `b` is produced by the first task, i.e., there is a *dependency* on memory location `b`. Note, however, that since the third task cannot be spawned until the first two return, memory location `c` is only accessed by the first task and `a` is only accessed by the second. So, any dependence analysis time spent checking for conflicts on `a` or `c` before the first two tasks start is unnecessary overhead, that delays the creation of the parallel tasks, possibly restricting available parallelism, and thus the scalability of the program. So, the `#pragma scoop task` directive spawning these tasks states that `c` and `a` are *safe* or *independent* arguments, that the analysis does not need to track. For the same reason, both the arguments of the third task are safe, meaning it can start to run without checking for dependencies.

2.3 Code Generation

The final SCOOP module transforms the input program to use BDDT for creating and managing tasks, disabling BDDT's runtime dependence checks for inferred or declared independent arguments. During the code generation phase, we make further improvements to the compiled code by producing custom code to interact with the dependence analysis and scheduler, instead of using the generic runtime's API. In particular, for each `#pragma scoop task` call, SCOOP generates a new

function that creates a task descriptor with the original function as task body, registers the task arguments with the runtime dependence analysis (BDDT only), and replaces the specified function call with the new task-creator wrapper function. Moreover, statically knowing the number of arguments and their data flow type, SCOOP is able to generate custom task invocation code that does not use loops to register task arguments with the runtime or expensive `va_arg` lists. Finally, SCOOP detects scalar arguments and automatically passes them by value, marking them as safe.

To demonstrate the code generation we use a derivation of Code 2.2, that requires implicit synchronization. Code 2.4 is an alternative way to express Code 2.1 using SCOOP annotations.

Code 2.4: Code 2.1 program written using SCOOP annotations, depending on implicit synchronization

```
1  int a = 1;
2  int b = 2;
3  int c = 3;
4  int d = 4;
5
6  void set(int *x, int *y) { *x = *y; }
7  void addto(int *x, int *y) { *x += *y; }
8
9  void main() {
10   #pragma scoop start(...)
11
12   #pragma scoop task inout(&a) in(&b)
13   addto(&a, &b);
14   #pragma scoop task inout(&c) in(&d)
15   addto(&c, &d);
16   #pragma scoop task inout(&a) in(&c)
17   addto(&a, &c);
18
19   #pragma scoop wait all
20
21   #pragma scoop task out(&b) in(&a)
22   set(&b, &a);
23   #pragma scoop task out(&c) in(&a)
24   set(&c, &a);
25   #pragma scoop task out(&d) in(&a)
26   set(&d, &a);
27
28   #pragma scoop finish
29 }
```

In this program the first two calls to `addto()` (lines 12–15) execute in parallel, but the third one (line 17), although annotated, will execute after the first two tasks because of the data dependencies on `a` and `c`. Then the program waits at line 19 until they all finish. After all tasks finish, the third call to `addto()` (line 17) is executed sequentially followed by the three parallel executions of `set()` (lines 16–26). Finally the program waits again for all tasks to finish at line 28.

In the rest of this section we discuss the SCOOP generated code for Code 2.4. Subsection 2.3.1 describes the code generation for x86 **S**hared **M**emory **P**rocessors (SMP) , while subsection 2.3.2 describes the code generation for the Cell processor. The main difference between the two architectures is that the Cell processor requires two separate executables, one for the **P**ower **P**rocessor **E**lement (*PPE*) and another for the **S**ynergistic **P**rocessing **E**lements (*SPE*), while x86 SMP architecture requires only one.

2.3.1 x86 SMP

In x86 SMP architectures the code generation is simpler, as we don't have to copy functions between files. The main transformations in these architectures are the creation of the custom functions that generate the task descriptors, the creation of the task table and the placement of the appropriate library calls for initialization, finalization and synchronization. Code 2.5 shows the SCOOP generated code for Code 2.4.

Code 2.5: SCOOP generated code for Code 2.4 targeting BDDT on x86

```
1  ...
2  void scoop_func_addto_0(int* x,int* y, int size0, int size1) {
3     ... //create task descriptor and pass to runtime
4    task_d->funcid = (uint8_t )0;
5
6    //1st argument
7    first_block = task_d->total_args;
8    AddAttribbute_Task(task_d, (void *)x, INOUT, arg_size0);
9    task_d->args[first_block ]. flag |= START;
10
11   //2nd argument
12   task_d->args[task_d->total_args].size = arg_size1;
13   task_d->args[task_d->total_args].address = (void*)y;
14   task_d->args[task_d->total_args].flag = INPUT|START|SAFE;
15   task_d->total_args++;
16    ...
17 }
18
19 void scoop_func_addto_1(int* x,int* y, int size0, int size1) {
20    ... //create task descriptor and pass to runtime
```

```
21    task_d->funcid = (uint8_t )0;
22
23    //1st argument
24    first_block = task_d->total_args;
25    AddAttribbute_Task(task_d, (void *)x, INOUT, arg_size0);
26    task_d->args[first_block].flag |= START;
27
28    //2nd argument
29    first_block = task_d->total_args;
30    AddAttribbute_Task(task_d, (void *)y, INOUT, arg_size1);
31    task_d->args[first_block].flag |= START;
32     ...
33  }
34
35  void scoop_func_set_0(int* x,int* y, int size0, int size1 ) {
36     ... //create task descriptor and pass to runtime
37    task_d->funcid = (uint8_t )1;
38
39    //1st argument
40    task_d->args[task_d->total_args].size = arg_size0;
41    task_d->args[task_d->total_args].address = (void*)x;
42    task_d->args[task_d->total_args].flag = OUTPUT|START|SAFE;
43    task_d->total_args++;
44
45    //2nd argument
46    task_d->args[task_d->total_args].size = arg_size1;
47    task_d->args[task_d->total_args].address = (void*)y;
48    task_d->args[task_d->total_args].flag = INPUT|START|SAFE;
49    task_d->total_args++;
50     ...
51  }
52   ...
53  void main(void) {
54    bddt_start (...);
55
56    scoop_func_addto_0(&a, &b, sizeof(int), sizeof(int));
57    scoop_func_addto_0(&c, &d, sizeof(int), sizeof(int));
58    scoop_func_addto_1(&a, &c, sizeof(int), sizeof(int));
59
60    bddt_wait_all();
61
62    scoop_func_set_0(&b, &a, sizeof(int), sizeof(int));
63    scoop_func_set_0(&c, &a, sizeof(int), sizeof(int));
64    scoop_func_set_0(&d, &a, sizeof(int), sizeof(int));
```

```
65
66     bddt_shutdown();
67 }
```

For `addto()`, SCOOP generated two different functions, `scoop_func_addto_0` (line 2) and `scoop_func_addto_1` (line 19). `scoop_func_addto_0()` handles the second argument as *independent*, excluding it from the dependence analysis graph , while `scoop_func_addto_1` handles both arguments as possible dependencies. SCOOP generated two different functions because in the lines 12–19 of Code 2.4, `b` and `d` can be considered *independent* as they don't alias and there is no data flow between them. SCOOP queries SDAM for this arguments and generates code accordingly. That said, the first two tasks (lines 56–57) can be spawned without the need to check their second argument for dependencies. However the third task (line 58) depends on the result of the previous two (lines 56–57), thus we use `scoop_func_addto_1()`, so that the runtime checks for its second argument in the dependency graph. Also notice that both `scoop_func_addto_0()` and `scoop_func_addto_1()` set the same `func_id`, that of `addto()`, to the task descriptor (lines 4 and 21).

For `set()`, SCOOP generated only one function, `scoop_func_set_0` (line 35). In the code segment between the lines 19 and 28 of Code 2.4, `a` is only being read and `b`, `c` and `d` do not alias, so the three `set()` calls can execute in parallel without any dependencies. `scoop_func_set_0()` issues all three `set()` calls, preventing the runtime from checking their arguments for dependencies.

2.3.2 Cell BE processor

As previously stated, the Cell processor requires the generation of two different executables. SCOOP takes as input a single `#pragma` annotated program's source code and creates two source code files as output, one for the PPE and the other for the SPUs.

The PPE file contains the whole program as well as the generated functions for the task issuing. Code 2.6 shows a part of the generated PPE file.

Code 2.6: SCOOP generated code for Code 2.4 targeting BDDT on the Cell processor

```
1  ...
2  void scoop_func_addto_0(int* x,int* y, int arg_size0, int arg_size1) {
3     ... //create task descriptor and pass to runtime
4    task_d->funcid = (uint8_t )0;
5
6    //1st argument
7    first_block = task_d->total_args;
8    DivideArgumentToBlocks(task_d, (void *)x, arg_size0, INOUT);
```

```
9     task_d->args[first_block].flag |= START;
10
11    //2nd argument
12    task_d->args[task_d->total_args].size = arg_size1;
13    task_d->args[task_d->total_args].flag = INPUT|START|SAFE;
14    task_d->args[task_d->total_args].address = (void*)y;
15    task_d->total_args++;
16     ...
17  }
18
19  void scoop_func_addto_1(int* x,int* y, int arg_size0, int arg_size1) {
20     ... //create task descriptor and pass to runtime
21    task_d->funcid = (uint8_t )0;
22
23    //1st argument
24     first_block = task_d->total_args;
25    DivideArgumentToBlocks(task_d, (void *)x, arg_size0, INOUT);
26    task_d->args[first_block].flag |= START;
27
28    //2nd argument
29     first_block = task_d->total_args;
30    DivideArgumentToBlocks(task_d, (void *)y, arg_size1, INPUT);
31    task_d->args[first_block].flag |= START;
32     ...
33  }
34
35  void scoop_func_set_0(int* x,int* y, int arg_size0, int arg_size1) {
36     ... //create task descriptor and pass to runtime
37    task_d->funcid = (uint8_t )1;
38
39    //1st argument
40    task_d->args[task_d->total_args].size = arg_size0;
41    task_d->args[task_d->total_args].flag = OUTPUT|START|SAFE;
42    task_d->args[task_d->total_args].address = (void*)x;
43    task_d->total_args++;
44
45    //2nd argument
46    task_d->args[task_d->total_args].size = arg_size1;
47    task_d->args[task_d->total_args].flag = INPUT|START|SAFE;
48    task_d->args[task_d->total_args].address = (void*)y;
49    task_d->total_args++;
50     ...
51  }
52   ...
```

```
53 void main(void) {
54   bddt_start (...);
55
56   scoop_func_addto_0(&a, &b, sizeof(int), sizeof(int));
57   scoop_func_addto_0(&c, &d, sizeof(int), sizeof(int));
58   scoop_func_addto_1(&a, &c, sizeof(int), sizeof(int));
59
60   bddt_wait_all();
61
62   scoop_func_set_0(&b, &a, sizeof(int), sizeof(int));
63   scoop_func_set_0(&c, &a, sizeof(int), sizeof(int));
64   scoop_func_set_0(&d, &a, sizeof(int), sizeof(int));
65
66   bddt_shutdown();
67 }
```

It is similar to Code 2.5, only with `DivideArgumentToBlocks` instead of `AddAttribbute_Task()`.

The SPU file contains copies of all the functions that have annotated calls, so the runtime can invoke them through the task dispatcher. Furthermore it contains the task dispatcher. Code 2.7 shows an example of a SCOOP generated BDDT task dispatcher for Code 2.2.

Code 2.7: SCOOP generated BDDT task dispatcher Code 2.4

```
1  int execute_task(queue_entry_t *ex_task, spe_task_state_t *task_info) {
2    int exit = 0;
3
4    switch(ex_task_d->funcid){
5      case 0:
6      {
7        set( (int *)(task_info->local[0]) ,(int *)(task_info->local[1]) );
8        break;
9      }
10     case 1:
11     {
12       addto( (int *)(task_info->local[0]) ,(int *)(task_info->local[1]) );
13       break;
14     }
15     default:
16     {
17       exit = 1;
18       break;
19     }
```

```
20    }
21
22    task_info->state = EXECUTED;
23    return exit;
24  }
```

2.3.3 Tiled arguments

SCOOP also supports tiles of arrays as arguments. To use such arguments the programmer must specify the array's number of columns, the tile's number of rows and the tile's number of rows. Code 2.8 shows an example where we want to process 25x50 tiles of an 100x100 array of integers. And Code 2.9 shows how the generated code by SCOOP looks like.

Code 2.8: Example of tiled arguments using SCOOP

```
1  void main(void) {
2    int int_array [100][100];
3     ...
4    arg1=&array[25][0];
5    arg2=&array[50][50];
6    #pragma scoop task in(arg1[25|50][100] ) out(arg2[25|50][100])
7     process_tile(arg1, arg2);
8     ...
9  }
```

Code 2.9: SCOOP generated code for Code 2.8

```
1   ...
2  scoop_process_tile_0(int* arg1, int* arg2,
3                       int els0, int el_sz0, int stride0,
4                       int els1, int el_sz1, int stride1){
5     ...
6    //1st argument
7    for(i=0; i<els0; ++i, arg1+=stride0){
8      first_block = task_d->total_args;
9      AddAttribbute_Task(task_d, (void *)arg1, INPUT, el_sz0);
10     task_d->args[first_block].flag |= START;
11   }
12
13   //2nd argument
14   for(i=0; i<els1; ++i, arg2+=stride1){
15     first_block = task_d->total_args;
16     AddAttribbute_Task(task_d, (void *)arg2, OUTPUT, el_sz1);
17     task_d->args[first_block].flag |= START;
```

```
18    }
19     ...
20  }
21   ...
22  void main(void) {
23    int int_array [100][100];
24     ...
25    arg1=&array[25][0];
26    arg2=&array[50][50];
27    scoop_process_tile_0(arg1, 25, 50*sizeof(int), 100*sizeof(int),
28                         arg2, 25, 50*sizeof(int), 100*sizeof(int));
29     ...
30  }
```

Each tile row is actually passed as a different argument, with size equal to tile columns × sizeof(int), to the runtime. This is done with the two loops at lines 7 and 14. In each loop iteration the address of the argument is ascended by the stride. The stride is calculated multiplying the array's number of columns with the size of its elements, in our case integers. Tiled arguments are pretty common in matrix manipulation and image processing algorithms.

2.4 Infrastructure

SCOOP first uses CIL [27], a front-end for the C programming language that facilitates program analysis and transformation, to parse and simplify the input program. SDAM then uses Locksmith's [11] points-to and data flow analysis engines to find aliasing among task arguments and implement the barrier analysis. Finally, SCOOP performs all code rewriting on the CIL AST, using the results computed by the SDAM analysis to tag any independent arguments for the runtime.

2.5 Regions

We have extended BDDT to support non-hierarchical regions. A region is like a stack of unbounded size which grows, until the region in its entirety is freed. The region library introduced provides three API calls, `bddt_newregion()`, `bddt_ralloc( region, size)` and `bddt_deleteregion( region )`. Regions are implemented as structures including their allocator and logistics about the allocated memory segments. As its name implies `bddt_newregion()` creates a new region. `bddt_ralloc( region, size)` allocates `size` bytes from `region` and returns the address pointing to them. Memory segments allocated using `bddt_ralloc( region, size)` are freed when `bddt_deleteregion( region )` is called. `bddt_deleteregion( region )` returns the freed memory segments to a memory pool so they can be used by other regions.

Regions enable BDDT to handle dynamic data structures as well as dynamic memory allocation within tasks. For example, a typical use of a region is to hold a list. All list nodes belong to the same region. This is achieved by assigning a region to the list and then allocating all the nodes of the list in this same region. This way we can add a whole list to the footprint of the task (using its region) , while without region support we should put all the node addresses to the task footprint. Furthermore region support enables tasks to manipulate dynamic data structures, such as lists, hashtables etc. With regions we can add a whole dynamic data structure to the footprint of a task (using its region). In contrast, without region support we should put all the node addresses to the task footprint, something not always possible and extremely hard.

When it comes to dynamic memory allocation inside tasks, the programmer must have previously added a region in the tasks footprint as `out` or `inout`. Then she can allocate as much memory as she needs with `bddt_ralloc()` from that region inside the task. This enables parallel manipulation of dynamic data structures. For example the initialization of a hashtable. After sequentially determining the bucket to insert the element a task can be spawned to push the element in the bucket list.

Furthermore, by their implementation BDDT regions provide better memory locality to dynamic data structures and reduced overheads for both memory management and dependence analysis. The region memory pools are allocated as pages and whenever `bddt_ralloc()` requests some space the region library gives the next free segment of the page or if that is not enough, which is not the common path, gives a segment of a new page. This way most of a list's nodes, for example, will be allocated in continuous memory, resulting in less memory misses on list traversals. Additionally allocating a whole page instead of each node separately, reduces the allocation cost with the trade of that some memory segments might be allocated but never used. The same is with free where instead of invoking free for each node, we free the whole region at once. Moreover, regions remove many dependency checks from BDDT. Each region is checked once without the need to check all of it's memory blocks.

Chapter 3

Evaluation

This chapter presents an evaluation of the SCOOP's transformations impact on the benchmarks' running time and scalability. Furthermore we explore additional ways for a programmer to expose more independent task arguments to SDAM, and how this can affect performance.

Also note, that we have placed barriers between the initialization process and the parallel section of the code. Although we do this to measure the applications executions time without interference from rogue tasks from the initialization process, some Read-after-Write dependencies are eliminated by SCOOP because of these barriers.

The evaluation was performed mostly on the x86 SMP architecture, due to the ease of deployment over the PS3. Furthermore the Cray node offers 24 cores over the PS3's 1 PPE and 6 SPEs. Allthoug the two platforms are completely different and the runtime developers where challenged by different aspects, we believe that our source to source transformations are not bound to the architecture. Thus we present results from both architectures but focus mainly on the x86 SMP architecture.

We observe that not all benchmarks offer opportunities to detect *safe* arguments. We show figures on the number of *safe* arguments found by SDAM and the number of *safe* arguments found by manually examining the code. Moreover, we show how we promote scalability with certain benchmarks by removing task arguments from the runtime analysis. Finally, we examine how we can expose more *safe* arguments with some extra effort from the programmer, how this can benefit performance and how well our tool does in detecting such safe arguments.

3.1 Evaluation Methodology

We evaluate SCOOP on a modest set of representative benchmarks, including several computational kernels and small-sized parallel applications on both architectures. This chapter describes the experimental setup, benchmarks, and methodology used for the evaluation presented in chapter 3.

3.1.1 Platforms

x86 SMP

We conduct all experiments for the x86 SMP architecture on one compute node of a Cray XE6 supercomputer with two AMD 2.1 GHz 12-core processors and 32GB of main memory. We also use one Intel Xeon E5520 2.27GHz, with 12GB of main memory for the single-core runs of Figure 1.1. All executables are generated on the Intel Xeon system with GCC 4.4.3 and the -O3 flag.

Cell BE processor

For the Cell architecture evaluation we used a sony PlayStation 3 (PS3) game console system, equipped with a 3.2 GHz Cell processor and 256 MBytes of main memory. On the PS3, applications are allowed to access only six out of the eight SPEs in the Cell processor. All benchmarks are first source to source compiled using SCOOP on an x86 machine and then compiled on the PS3 with GCC 4.1.2 and the -O3 flag.

3.1.2 Benchmarks

We use seven benchmarks for the x86 SMP evaluation and four more for the Cell BE processor evaluation. Next, we briefly discuss each benchmark.

x86 SMP

Black-Scholes is a parallel implementation of a mathematical model for price variations in financial markets with derivative investment instruments. It decomposes and processes the data in rows. This Black-Scholes implementation is taken from the PARSEC [32] benchmark suite.

SMPSs-FFT is an implementation of a 2 dimensional Fast Fourier Transform algorithm. This FFT implementation is part of the SMPSs distribution, and consists of five parallel loops that alternate in transposing the input array and performing 1 dimensional FFT on each row. Each task created in the FFT calculation loop operates on an entire row of the array, while transposition phases break the array into tiles and create a task to transpose a group of tiles.

SPLASH-FFT is an alternative FFT kernel implementing a similar 2 dimensional algorithm. SPLASH-FFT is part of the SPLASH-2 [33] benchmark suite.

GMRES is an implementation of the iterative Generalized Minimal Residual method for solving systems of linear equations. GMRES decomposes and processes the input data in array rows.

Jacobi is a parallel implementation of the Jacobi method for solving systems of linear equations. It uses a 2-dimensional array with tiled layout. This Jacobi implementation is part of the SMPSs distribution. Each parallel task in Jacobi processes a tile of the array with a kernel implementing a 5-point stencil computation.

HPL solves a random dense linear system in double precision arithmetic. It is part of the High-Performance Linpack Benchmark.

Multisort is a parallel implementation of Mergesort. Multisort is an alternative implementation of the Cilksort [1] example, and has two phases: during the first phase, it divides the data into chunks and sorts each chunk. During the second phase, it merges those chunks.

Cell BE processor

Cholesky is the blocked sparse Cholesky factorization kernel from SPLASH-2 [33]. It factors a sparse matrix into the product of a lower triangular matrix and its transpose. It is similar in structure and partitioning to the LU factorization kernel (see below), but has two major differences: (i) it operates on sparse matrices, which have a larger communication to computation ratio for comparable problem sizes, and (ii) it is not globally synchronized between steps.

LU kernel is also taken from SPLASH-2. It factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense n x n matrix A is divided into an N x N array of B x B blocks (n = NB) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The block size B should be large enough to keep the cache miss rate low, and small enough to maintain good load balance. Fairly small block sizes (B=8 or B=l 6) strike a good balance in practice. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to processors that own them. See [34] for more details.

SAXPY stands for **S**ingle-precision real **A**lpha **X** **P**lus **Y**. SAXPY is one of the first level functions in the Basic Linear Algebra Subprograms (BLAS) package. SAXPY is a combination of scalar multiplication and vector addition, $z = \alpha x + y$, where α is a scalar and x and y are vectors. [35]

SGEMV computes the matrix-vector product for either a real general matrix or its transpose, using the scalars α and β, vectors x and y, and matrix A or its transpose A^T:

$$z = \beta y + \alpha A x$$

or

$$z = \beta y + \alpha A^T x$$

3.1.3 Measurement Methodology

In our experiments, we measure the performance of the parallel section of the code, excluding any initialization and I/O at the start and end of each benchmark. In some of the benchmarks, initialization is also done in parallel to achieve locality in NUMA machines, although it is not optimized for performance or counted for running times.

For each benchmark, we ran a *basecase* version we wrote manually, using the BDDT API to spawn tasks. The base case does not use any annotations for independencies, hence BDDT dynamically tracks all task arguments for dependencies.

	Benchmark	**LOC**	**Tasks**	**Total Args**	**Scalar Args**
x86 SMP	Black-Scholes	1540	1	8	1
	SMPSs-FFT	2147	8	36	25
	SPLASH-FFT	2920	4	12	0
	GMRES	2652	18	72	20
	HPL	2396	11	63	35
	Jacobi	1076	1	6	0
	Multisort	1118	3	8	4
Cell BE	Cholesky	2195	4	8	0
	LU	2819	3	10	3
	SAXPY	1675	1	3	1
	SGEMV	2159	1	4	1

Table 3.1: Benchmarks' characteristics

Table 3.1 shows the size of each benchmark in lines of code [1] , the number of task invocations, the total number of arguments of all tasks, and how many of those arguments are scalars. All scalar arguments are marked as safe in both version of the code for each benchmark, since it is trivial for either the programmer or the analysis to find them, thus we do not count them as *safe* arguments discovered by the analysis.

3.2 Overhead Reduction

Table 3.2 shows the effect of SCOOP on the total running time of all benchmarks, listed in the first column, when run on 24 cores on the Crey node and on 6 SPEs on the PS3. The second column shows the total running time in milliseconds when using BDDT to perform runtime dependency analysis on all task arguments of all tasks. The third column shows the total running time in milliseconds for the benchmark compiled with SCOOP, without runtime dependence checking for arguments

[1] We count lines of code after preprocessing and merging all source files, not counting blank lines and comments

found independent by the analysis. The fourth column shows the relative speedup percentage [2] gained by applying SCOOP, and the last two columns show the total number of task arguments found to be independent, out of the total number of non-scalar task arguments in each program.

	Benchmarks	**BDDT (msec)**	**SCOOP (msec)**	**Speedup%**	**Inferred Args**	**Non Scalar Args**
x86 SMP	Black-Scholes	452.07	138.75	69.31	7	7
	SMPSs-FFT	526.59	524.37	0.42	0	11
	SPLASH-FFT	903.05	698.87	22.61	7	12
	GMRES	4610.88	4379.04	5.03	9	52
	HPL	612.82	604.43	1.37	1	28
	Jacobi	3518.36	3503.88	0.41	0	6
	Multisort	4098.82	4100.36	-0.04	0	4
Cell BE	Cholesky	19.37	19.42	-0.26	0	8
	LU	9172.66	9046.37	1.37	3	7
	SAXPY	248.91	243.51	2.16	2	2
	SGEMV	27.55	23.41	15.03	2	3

Table 3.2: Running times and speedup using 24 cores on x86 SMP and 1 PPE + 6 SPEs on the Cell BE

Overall, the average speedup from applying SCOOP on all benchmarks is 10.67%. This value however, is not representative, since the actual impact greatly varies among benchmarks. Specifically, the independence analysis is able to infer safe task arguments only in six out of the eleven benchmarks. This has a large impact on the overhead and scalability of the BDDT runtime dependence analysis, producing substantial speedup over the original BDDT versions. On the other hand, the analysis did not discover any safe task arguments in SMPSs-FFT, Jacobi, Multisort, or Cholesky, which do not gain any significant benefit from SCOOP, other than that resulting from custom code generation (avoiding `va_args` in the BDDT API, etc.). We consider the slowdowns in Multisort and Cholesky to be within noise.

It is noticeable that allthough SCOOP completely removes the dynamic dependence analysis for SAXPY it doesn't get the expected speedup. This reveals that in the Cell BE we don't achieve more parallelism removing dependencies, probably, due to the low number of workers. Figure 3.1 shows the breakdown of the Cell BE execution times. We only present the three benchmarks where SCOOP was able to infer some *safe* arguments. *ADAM* are the breakdowns of the benchmarks written using BDDT's API. *SCOOP* are the breakdowns of the benchmarks written using SCOOP annotations and compiled after a SCOOP pass with SDAM disabled. Finally *SDAM* are the breakdowns of the benchmarks written using SCOOP an-

[2] The speedup percentage is calculated as $\frac{time_{BDDT} - time_{SCOOP}}{time_{BDDT}} \times 100$

notations and compiled after a SCOOP pass with SDAM enabled.

The breakdown in details:

- *Complete time* is the time required to update the dependency graph for each completed task.
- *Issue time* is the time needed by the runtime to check for dependencies and send the tasks to the workers.
- *Instantiate time* is the time needed by the runtime to create the tasks' descriptions and their arguments' descriptions, as well as to add them in *DAG* (Dependence Analysis Graph).
- *Stall time* is the time spent polling the worker queues for an available slot when the master has tasks eligible for execution.

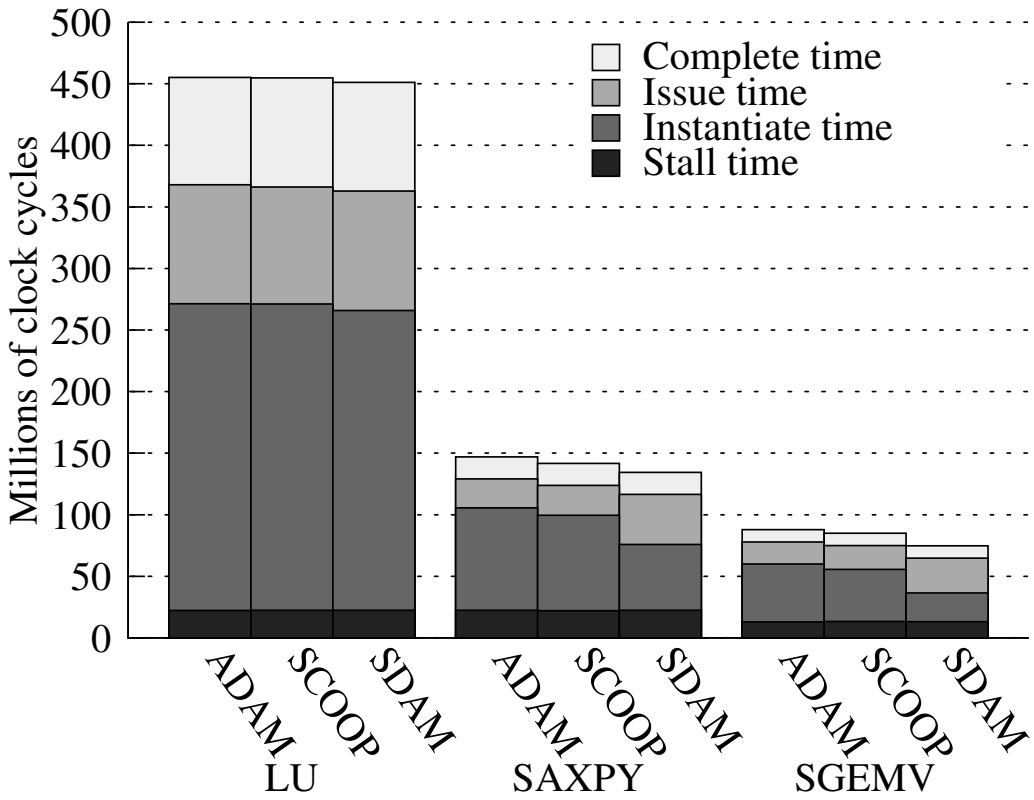


Figure 3.1: Breakdown of the Cell BE execution times

As shown in Figure 3.1, instantiation time is reduced by 2%-50% by excluding arguments from the runtime analysis in addition to the benefit from SCOOP's optimizations. The benefit of the analysis depends on the complexity of the code

and the number of *safe* arguments it can detect. This benefit however is countered to some extent by a rise in issue time. Overall we get 1%-17% and 0.1%-5% speedup from SCOOP over the manually written code, with and without SDAM support respectively.

3.3 Scalability

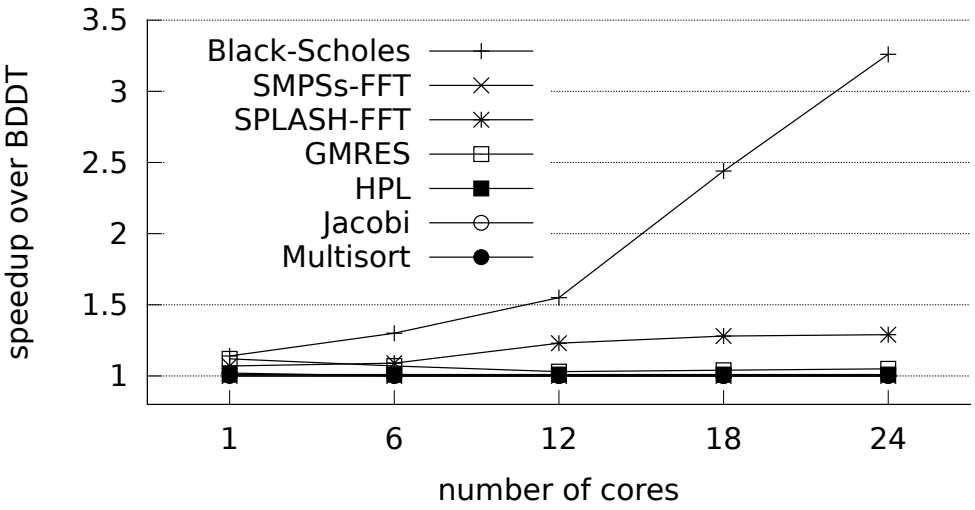


Figure 3.2: Scalability of SCOOP-generated code on x86 SMP architecture

Figure 3.2 shows the speedup each benchmark gains from SCOOP inference of safe arguments, compared to dynamic dependence analysis of all arguments, with respect to the number of cores executing the program. For instance, note that Black-Scholes, when executed on a single core, does not benefit considerably from the SCOOP independence analysis. As the core count grows, however, the difference in overhead in performing all dynamic dependency checks on the safe arguments becomes significant.

3.4 Exposing Independencies

In some of our benchmarks, the original code of SMPSs-FFT, Jacobi, HPL and Multisort did not offer any *safe* arguments. We tried to place barriers on certain program points in order to remove some dependencies across loops that contained tasks and create more *safe* arguments. This method allowed us to manually find a great deal more safe arguments on a couple of benchmarks, namely SMPSs-FFT, SPLASH-FFT, Jacobi and HPL. On the other hand, placing barriers did not help us find any *safe* arguments on Multisort, since Multisort does not have any independencies as the second phase of the algorithm uses the output data of

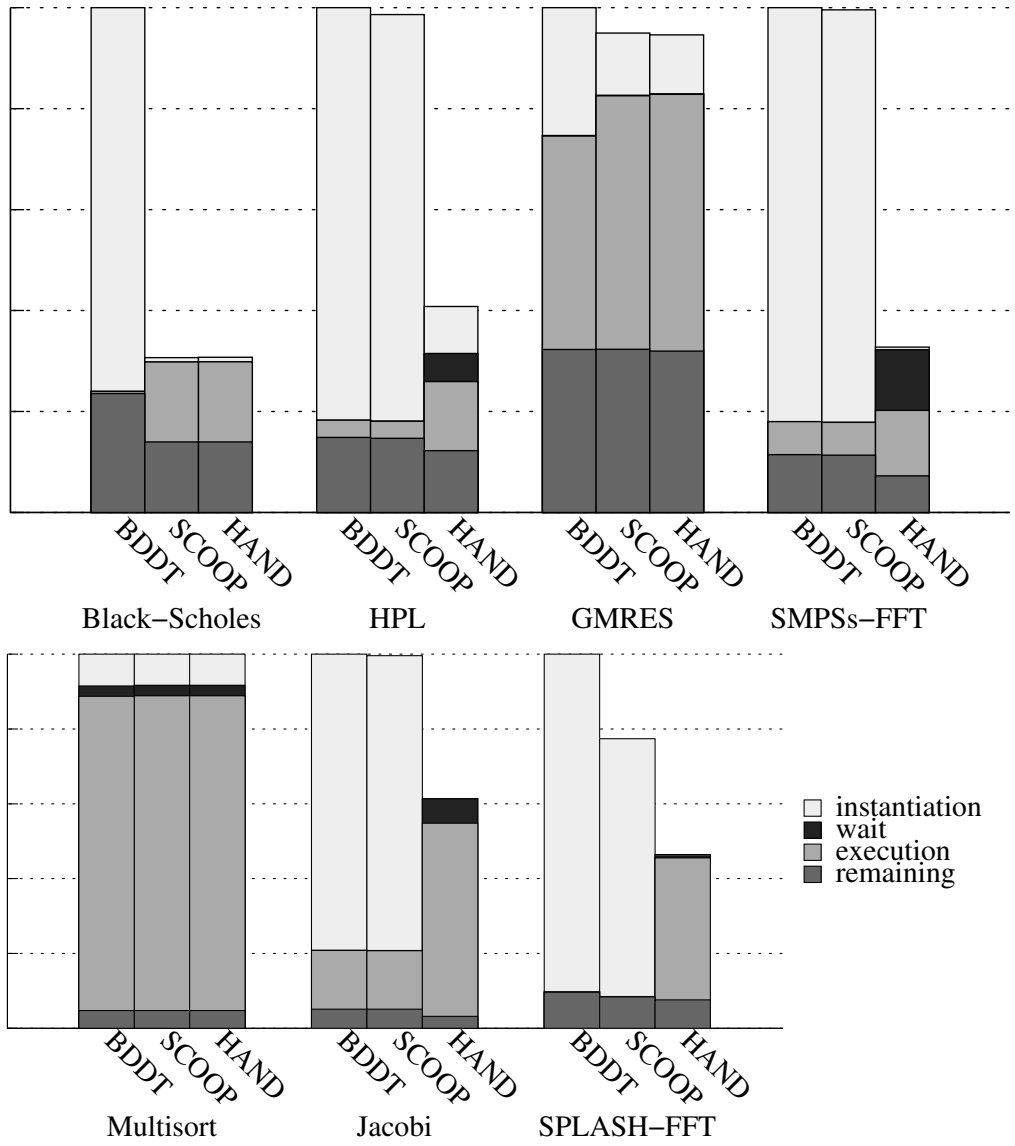


Figure 3.3: Comparison between BDDT, SCOOP and hand tuned versions of the code. on y axis is the normalized execution time

the first one as input, implying RAW dependencies. Furthermore, Multisort as a recursive application has a lot of do-across parallelism and does not benefit from barriers placement. Another problem for SCOOP is that although we exposed more independencies in the code, the analysis failed to detect them, due to it's inability to handle complex array indexes in loops and eliminate self dependencies of out/inout task arguments. For this reason we wrote a third version of the code, adding barriers and safe arguments manually, using the *safe...* annotation. In Figure 3.3 we present normalized breakdowns of the benchmarks' execution times.

The breakdown in details:

- *instantiation*, which is the time needed by the runtime to create a task description and its arguments' descriptions, as well as to add it in *DAG* (Dependence Analysis Graph).
- *wait*, which is the time spent waiting at a barrier.
- *execution*, which is the time spent executing tasks at the Master due to the execution window of *DAG* being full.
- *remaining*, which is the remaining execution time including the time spent for running the sequential parts of the code.

Figure 3.3 shows how the hand tuned code performs compared to BDDT and SCOOP versions of the code. The hand tuned version outperforms both other versions in the cases that barriers exposed more *safe* arguments. For the cases that we do not find more *safe* arguments by adding barriers, SCOOP's code and the hand tuned versions are identical, thus we observe no benefit. Note that, although GMRES gains nothing by adding barriers, in the hand tuned version we marked the arguments that the analysis had missed as *safe* , in order to get some speedup.

3.5 Extended evaluation of x86 SMP architecture

In section 3.4 we explore ways to expose more independencies. In this chapter we present the extensive exploration we made to achieve our current understanding. As described in section 3.4, placing barriers brings forth more independencies. Finding those independent arguments and marking them as *safe* yields great performance improvement, with an average of 57.57%. However this is not always possible and depends on the benchmarks nature. Benchmarks with tasks in loops are more likely to benefit from this technique, while applications with tasks in recursive functions, like Multisort, or with excessive do-across parallelism , like GMRES, don't seam to be able to benefit from barriers.

In the rest of this section we compare several versions of the x86 benchmarks. First of all we have *BASE*, which is the application written using the BDDT's API. We also run another version of the code using SCOOP's `#pragma` directives and compiled after a SCOOP pass, we call this *SB* (Scoop compiled Basecase).

For the benchmarks we are able to detect safe arguments examining the code or using SDAM, we ran two more versions, *SBS* (Scoop compiled Basecase with SDAM) and *ASA* (Scoop compiled with Annotated Safe Args). *SBS* is the same code with SB, but in this version we use SDAM to find independencies and generate appropriate code. *ASA* is written using SCOOP's `#pragma` directives and the `safe(...)` notation to manually mark safe arguments. When SCOOP passes *ASA*, SDAM is disabled.

Additionally, for the benchmarks we are able to get better performance adding barriers to imply more safe arguments, we run two more versions, *SABS* (Scoop compiled Annotated with Barriers and using Sdam) and *BE* (Scoop compiled programmers Best Effort). *SABS* is the same code with *SBS*, but with some barriers placed in key points to create more independencies. In *SABS* we use SDAM to find those extra independencies. *BE* is the same as *SABS* but in this version we use the `safe(...)` notation to mark the extra safe arguments implied from the barriers. When SCOOP passes *BE*, SDAM is disabled.

When a benchmark doesn't have the *SBS* and *ASA* versions, it means that we were not able to find any safe arguments manually nor with SDAM. When a benchmark doesn't have the SABS and BE versions, it means that we were not able to get any performance improvement placing barriers in order to have more safe args.

Table 3.3 sums up the properties of each code version. Column *SCOOP* means that this versions passes from SCOOP. Column *SDAM* means that SDAM is enabled during the SCOOP pass. Column *SAFE ARGS* indicates whether `safe(...)` notation is used to manually mark *safe* arguments. Finally *BARRIERS* column stands for whether we introduced extra barriers in order to expose independencies.

Initialism	**SCOOP**	**SDAM**	**SAFE ARGS**	**BARRIERS**
BASE	✗	✗	✗	✗
SB	✓	✗	✗	✗
SBS	✓	✓	✗	✗
ASA	✓	✓	✓	✗
SABS	✓	✓	✗	✓
BE	✓	✓	✓	✓

Table 3.3: Properties of the different code versions

Running SDAM on the benchmarks with extra barriers does not remove any dependencies from our benchmarks, thus we omit the SABS bar from the graphs. This is happening because, although barriers usually enable loops with tasks to become do-all, SDAM fails to mark all the arguments of a task in a loop as *safe*, due to analysis limitations, which are beside the scope of this work. However, as mentioned earlier, we place barriers in all versions of the benchmarks between the initialization process and the parallel section, in order to measure the later without any interference from rogue tasks of the initialization. Because of these barriers SDAM is able to eliminate some RAW dependencies.

In our evaluation we exclude scalar variables. From this point on, when referring to *safe* arguments, we only consider pointers to the actual data of each benchmark, since it is trivial for both the analysis and the programmer to detect the former

and their impact on reducing the runtime overhead, which is marginal.

In this section we mainly focus on the performance improvement on 24 cores and on the total behavior of the different benchmarks versions over different number of cores.

Plot explanation: On the y-axis is the execution time in milliseconds. On the x-axis is the number of cores. Each bar shows the breakdown of a benchmark's version (described above).

Master's breakdown:

- *instantiate time* is the time needed by the runtime to create a task description and it's arguments' descriptions, as well as to add it in *DAG* (Dependence Analysis Graph).
- *wait time* is the time spent waiting at a barrier.
- *stall/exec time* is the time spent executing tasks at the Master due to the execution window being full.
- *rest time* is the time spent for running the sequential parts of the code.

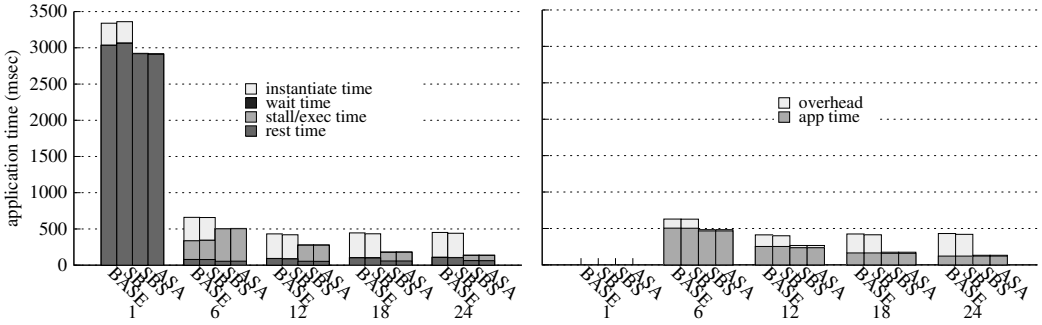
Worker's breakdown: For the workers we show an indicative but not exact breakdown of the execution time to runtime overhead and application time. As there where no significant variations we present the average breakdown.

- *overhead* is the time spent for the runtime's mechanisms. Includes task stealing, task dequeue and task completion (release dependencies).
- *app time* is the time spent executing actual application's code.

The formula that calculates the actual application time is:
stall/exec time+*rest time*+$(N-1)\times$ *app time*
where N is the number of cores.

3.5.1 Black-Scholes

Figure 3.4 shows execution times for BDDT (*BASE*), SCOOP base (*SB*), SCOOP base with SDAM running (*SBS*) and SCOOP with manually annotating *safe* args (*ASA*). We get 2.58% performance improvement from the compiler code generation. We also observe 69.3% benefit in performance for both *SBS* and *ASA*. This is due to SDAM finding all the *safe* arguments found manually by examining the code. Black-Scholes shows that removing overheads can improve the scaling factor of an application. In more detail in Figure 3.4 we see that Black-Scholes without its *safe* arguments marked scales up 12 cores. On the other hand, after running it with the *safe* arguments marked we see that it gradually scales till 24 cores.

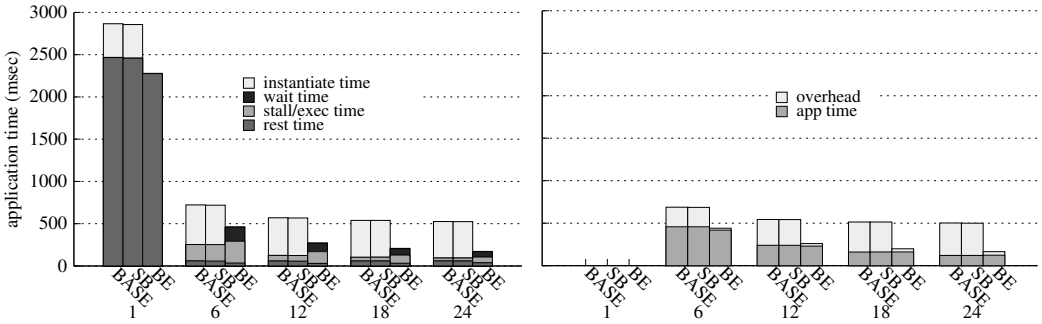


(a) Master's breakdown (b) Worker's breakdown

Figure 3.4: Black-Scholes Evaluation

3.5.2 SMPSs-FFT

Figure 3.5 shows execution times for the BDDT (*BASE*), SCOOP base (*SB*) and SCOOP with manually annotated *safe* arguments and barriers (*BE*). FFT compiler generated code (*SB*) gives insignificant speedup. Moreover, SDAM fails to find any *safe* arguments, even if we add barriers, due to its inability to eliminate self dependencies , thus bars *SABS* and *SBS* were omitted. We manage to find ten *safe* arguments by manually tuning the code using barriers, which results in a speedup of 67.25%.



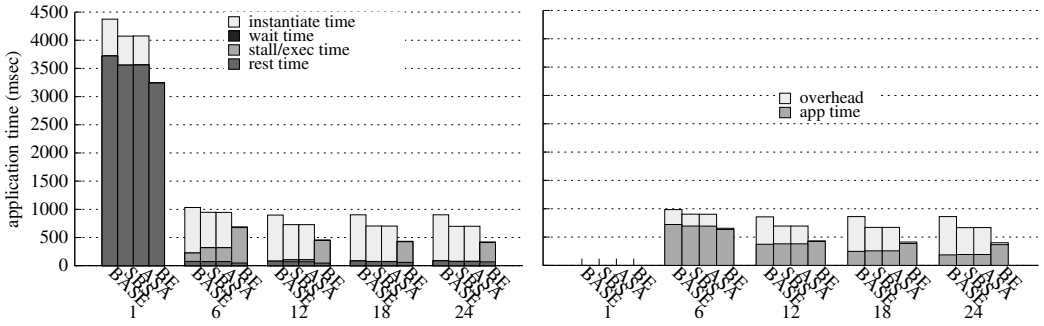
(a) Master's breakdown (b) Worker's breakdown

Figure 3.5: SMPSs-FFT Evaluation

3.5.3 SPLASH-FFT

Figure 3.6 shows execution times for BDDT (*BASE*), SCOOP base (*SB*), SCOOP base with SDAM running (*SBS*), SCOOP with manually annotating *safe* arguments (*ASA*) and SCOOP with manually annotated *safe* arguments and barriers (*BE*). FFT(SPLASH) compiler generated code (*SB*) gives insignificant speedup,

thus it was omitted from the plot. In *SBS* version, however, the analysis finds all the *safe* arguments we found manually, resulting in a speedup of 22.61%. By adding barriers to the code the analysis failed to find any additional *safe* arguments. However, we managed to manually discover more *safe* arguments, achieving a speedup of 53.55% when using the hand tuned code (*BE*).



(a) Master's breakdown (b) Worker's breakdown

Figure 3.6: SPLASH-FFT Evaluation

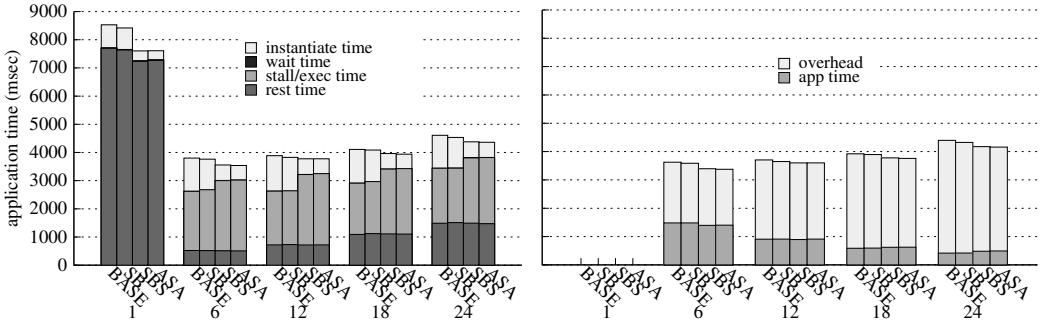
3.5.4 GMRES

Figure 3.7 shows execution times for BDDT (*BASE*), SCOOP base(*SB*), SCOOP base with SDAM running (*SBS*) and SCOOP with manually annotating safe args (*ASA*). We get 1.68% performance improvement from the compiler code generation. We also observe 5.03% benefit in performance for *SBS*, while for *ASA* the corresponding benefit is 5.4%. This is due to SDAM finding eight out of the eleven safe arguments found manually examining the code.

Figure 3.7 plots the numbers of a GMRES execution with 512 block size. Using a 1024 block size reduces the previous numbers from 1.68% for *SB* to 0%. And for *SBS* and *ASA* versions the 5.03% and 5.4% performance improvement becomes 0.6% and 1.82% respectively. Such results show that small block sizes introduce greater overheads to the runtime, rendering it less flexible considering the task data granularity. SCOOP makes the programming model even more flexible by removing those overheads.

3.5.5 Jacobi

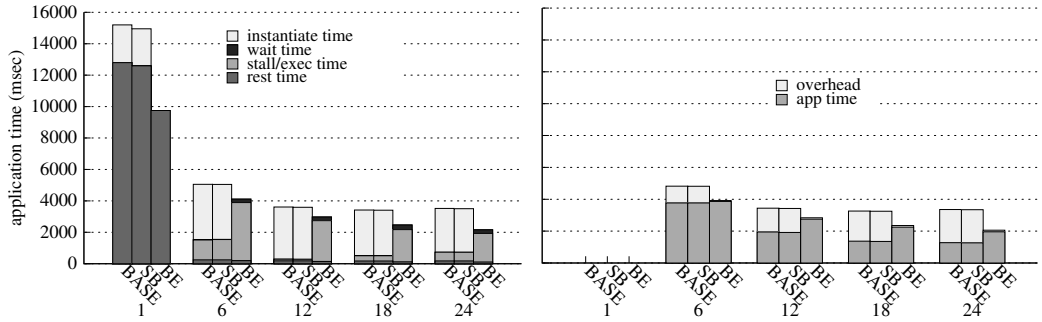
Figure 3.8 shows execution times for BDDT (*BASE*), SCOOP base (*SB*) and SCOOP with manually annotated *safe* args and barriers (*BE*). Jacobi compiler generated code (*SB*) gives insignificant speedup. We are not able to find any *safe* arguments neither examining the code nor with SDAM (so *SBS* and *ASA* bars are omitted). We only manage to find *safe* arguments by placing a barrier after the main loop of the benchmark, which transforms the inner loop to a do-all



(a) Master's breakdown (b) Worker's breakdown

Figure 3.7: GMRES Evaluation

loop. We observe a speedup of 38.61% due to the manually placed annotations (*BE* bar). SDAM fails to detect the *safe* arguments becoming available after the barrier placement due to the nested loop accessing blocks in non-sequential order.

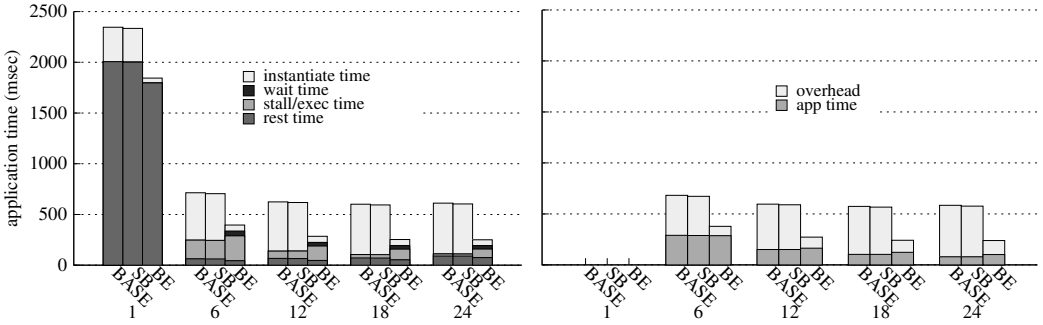


(a) Master's breakdown (b) Worker's breakdown

Figure 3.8: Jacobi Evaluation

3.5.6 HPL

Figure 3.9 shows execution times for BDDT (*BASE*), SCOOP base (*SB*) and SCOOP with manually annotated *safe* args and barriers (*BE*). We observe a 1.37% speedup from the compiler generated code (*SB* bar). We did not find any *safe* arguments neither examining the code nor using SDAM. However, by adding barriers, we managed to find some *safe* arguments and get a performance improvement of 59.19% (*BE* bar). SDAM fails to detect the *safe* arguments becoming available after the barrier placement due to complex indexing of the arrays.

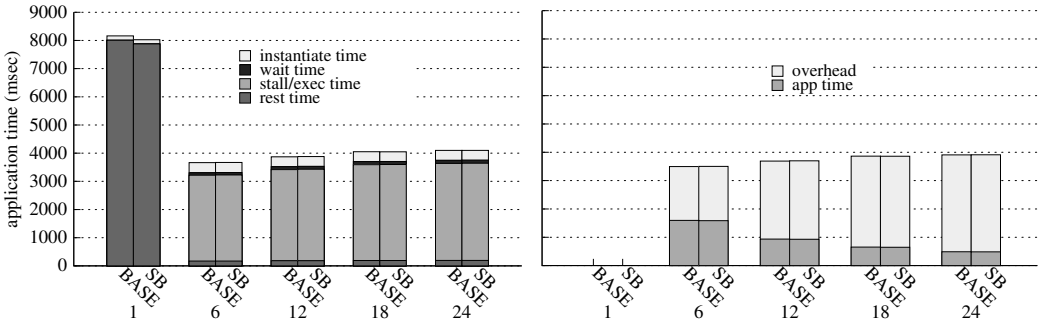


(a) Master's breakdown (b) Worker's breakdown

Figure 3.9: HPL Evaluation

3.5.7 Multisort

Figure 3.10 shows execution times for BDDT (BASE) and SCOOP base(SB). Multisort compiler generated code (*SB*) gives insignificant speedup. We can not exploit any *safe* arguments in Multisort neither using SDAM nor manually examining the code, even with barrier placement. Multisort does not have any independencies because the second phase of the algorithm uses the output data of the first one as input, implying RAW dependencies. Furthermore Multisort, as a recursive application, has a lot of do-across parallelism and does not benefit from barriers placement.



(a) Master's breakdown (b) Worker's breakdown

Figure 3.10: Multisort Evaluation

It's noticeable that the reduction of *issue time* implies an increase in *stall/exec time*. This is expected in BDDT's architecture. With faster instantiation times BDDT's task execution window gets full earlier and the master thread starts executing tasks in an earlier phase, enabling more parallelism. Also, examining the breakdowns we can see that the *app time* is inversely proportional to the *stall/exec time*. This is reasonable, because when the Master executes work, it offloads the

workers. However in SPLASH-FFT, Jacobi and GMRES, as mentioned before the workers' execution breakdown is indicative.

Our evaluation shows that SCOOP's optimizations manage to reduce overhead on some applications, however our hand tuned version of the code shows that there is room for improvement by using a more precise analysis. Moreover, we manage to get a substantial speedup in some benchmarks and we believe that using techniques to remove *safe* arguments from the runtime analysis, can have an important impact on reducing overheads.

Chapter 4

Conclusions and Future Work

We present SCOOP, a compiler for a task-parallel extension of C with implicit synchronization. SCOOP targets BDDT, a runtime system that uses dynamic dependence analysis to automatically synchronize and schedule parallel tasks. SCOOP uses static analysis to infer safe task arguments and reduce the overhead for tracking accesses to that memory in BDDT. We have tested SCOOP on a set of parallel benchmarks, in which it finds and removes unnecessary runtime checks on task arguments.

SCOOP was able to infer safe task arguments in seven out of the eleven benchmarks. We compared our results of the static analysis to manually examining the code and marking indepdendent arguments. Our results show that the hand tuned code in most cases performed better than the SCOOP generated. However, on certain cases, the static analysis managed to match the hand tuned code. We believe that there is potential in our method to reduce runtime overhead, by removing independent arguments.

Our experience taught us that in order to achieve the best results, the runtime team must cooperate with the compiler team and co-develop parts of both systems. This way the compiler team earns a better understanding of the possible problems and bottlenecks of the runtime, while the runtime team can ask for extra features and understand the limitations of the compiler.

SCOOP was initiated as an optimization tool. However our exprerince, using SCOOP and writing benchmarks both with BDDT's API and SCOOP annotations showed us that SCOOP not only can optimize the resulting program, but can also help the programmer during the development. Writing benchmarks with the BDDT's API is a lot harder than doing the same job with SCOOP annotations. In our latest work, when porting benchmarks, we first wrote them with SCOOP annotations. Then after debugging them, we transformed them from SCOOP annotations to BDDT's API calls to create the basecase. SCOOP with some extra effort can report a few possible errors to the programmer, such as wrong data flow annotation. A possible scenario would be a function that performs a write access

on one of its arguments and the programmer by mistake annotated this argument as *IN*. In this case SCOOP could easily print a warning.

Another possible alternative use of SCOOP would be to use it as a consultant for task footprints. SCOOP and SDAM with some extension will be able to provide the programmer with safe but in many cases conservative task footprints. Then the programmer will be able to have a working application using these footprints and with some effort remove the conservative annotations and make the application perform faster.

We currently focus on the region support and some benchmarks using it. As we discussed in section 2.5, regions introduce great flexibility to task-parallel programming models. With hierarchical regions and nested tasks we would be able to replicate the Sequoia model, in some extend. Enabling the use of recursion in BDDT. Our next target will probably be porting SCOOP to SCC or Formic [36]. SCC's code generation should be close to the CELL BE's one, as both architectures use the message passing programming model.

Bibliography

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *PPOPP*, 1995, pp. 207–216.

[2] *OpenMP Application Program Interface, Version 2.5*, OpenMP Architecture Review Board, May 2005.

[3] J. C. Jenista, Y. H. Eom, and B. Demsky, "OoOJava: Software out-of-order execution."

[4] R. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel Java."

[5] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword, "Synchronization via scheduling: Techniques for efficiently managing shared state."

[6] G. Tzenakis, A. Papatriantafilou, P. Pratikakis, J. Kessapides, H. Vandierendonck, and D. S. Nikolopoulos, "BDDT: Block-level Dynamic Dependence Analysis for Deterministic Task-Based Parallelism." Foundation for Research and Technology Hellas (FORTH) - Institute of Computer Science (ICS), Tech. Rep. TR-426, February 2012.

[7] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures."

[8] G. Tzenakis, K. Kapelonis, M. Alvanos, K. Koukos, D. S. Nikolopoulos, and A. Bilas, "Tagged Procedure Calls (TPC): Efficient runtime support for task-based parallelism on the cell processor," in *HiPEAC*, ser. Lecture Notes in Computer Science, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., vol. 5952. Springer, 2010, pp. 307–321.

[9] F. S. Zakkak, "Source to source compiler (S2S): C source level extensions and optimizations of parallelism on the Cell Processor." BSc Thesis, University of Crete, Heraklion, Crete, Greece, August 2010.

[10] D. Chasapis, "Static dependence analysis module for the s2s compiler," BSc Thesis, University of Crete, Heraklion, Crete, Greece, February 2011.

[11] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," *SIGPLAN Not.*, vol. 41, pp. 320–331, June 2006. [Online]. Available: http://doi.acm.org/10.1145/1133255.1134019

[12] M. Tofte and J.-P. Talpin, "Region-based memory management," *Inf. Comput.*, vol. 132, pp. 109–176, February 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=249657.249661

[13] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "Cellss: Making it easier to program the cell broadband engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593 –604, 2007.

[14] *SMP Superscalar (SMPSs) v2.3 User's Manual*, Barcelona Supercomputing Center, 2010.

[15] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[16] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, 2006, p. 4.

[17] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in cilk programs," in *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, 1997.

[18] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures."

[19] S. Cherem, T. Chilimbi, and S. Gulwani, "Inferring locks for atomic sections."

[20] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems.* New York, NY, USA: ACM, 2002.

[21] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman, "Safe nondeterminism in a deterministic-by-default parallel language."

[22] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August, "Commutative set: A language extension for implicit parallel programming."

[23] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," *Communications of the ACM*, vol. 8, pp. 4–13, 1992.

[24] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis."

[25] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection."

[26] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Practical static race detection for C," *ACM Trans. Program. Lang. Syst.*, vol. 33, pp. 3:1–3:55, January 2011.

[27] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *In International Conference on Compiler Construction*, 2002, pp. 213–228.

[28] J. Rehof and M. Fähndrich, "Type-base flow analysis: from polymorphic subtyping to cfl-reachability."

[29] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability."

[30] M. E. Wolf, "Improving locality and parallelism in nested loops," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1992.

[31] M. Wolfe, *High performance compilers for parallel computing.* Addison-Wesley, 1996.

[32] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[34] S. C. Woo, J. P. Singh, and J. L. Hennessy, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VI. New York, NY, USA: ACM, 1994, pp. 219–229. [Online]. Available: http://doi.acm.org/10.1145/195473.195547

[35] "SAXPY description on wikipedia." [Online]. Available: http://en.wikipedia.org/wiki/SAXPY

[36] "The Formic board from Crete, for building FPGA prototypes of Manycore Processors." [Online]. Available: http://www.alphagalileo.org/ViewItem.aspx?ItemId=109640&CultureCode=en